ESI Report Nr. 2008–10528

Tilt-tray Sorters modelled with UPPAAL

Roelof Hamberg

Document status: Final
Document release date: 25 March 2008

Distribution list

<table>
<thead>
<tr>
<th>Commentators</th>
<th>Copy-holders</th>
<th>Archive</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jozef Hooman, ESI</td>
<td>Koen Evers, Vanderlande</td>
<td>ESI archive</td>
</tr>
<tr>
<td>Jeroen Voeten, ESI</td>
<td></td>
<td>Falcon project archive</td>
</tr>
<tr>
<td>Frits Vaandrager, RUN</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Joris van den Aker, ESI</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Version</td>
<td>Date</td>
<td>Description of changes</td>
</tr>
<tr>
<td>---------</td>
<td>--------------------</td>
<td>---------------------------------</td>
</tr>
<tr>
<td>0.5</td>
<td>25 January 2008</td>
<td>Creation of document</td>
</tr>
<tr>
<td>0.7</td>
<td>12 February 2008</td>
<td>Review &amp; Update</td>
</tr>
<tr>
<td>0.9</td>
<td>6 March 2008</td>
<td>Review &amp; Update</td>
</tr>
<tr>
<td>1.0</td>
<td>25 March 2008</td>
<td>Adapt to ESI style &amp; Finalize</td>
</tr>
</tbody>
</table>
Abstract

This is a report of modeling activities done in the second half of an internship at the Radboud University in Nijmegen in the group of Frits Vaandrager. The main goal was to become acquainted with the possibilities and limitations of timed automata, model verifications, and one of the leading tools in this area, UPPAAL.

Tilt-tray sorters were modelled with UPPAAL in order to be able to improve on the heuristic scheduling rules of their input connections. This goal has not been achieved. Nevertheless, this research has been useful because the presented approach appears feasible and promising when the practical problems with UPPAAL have been solved.
Contents

Abstract ................................................................. 3
1 Introduction ......................................................... 5
2 Problem Statement .................................................. 5
3 Approach ............................................................... 7
4 Evolution of Models .................................................. 7
   4.1 Package Generation and Exits ................................. 8
   4.2 Induct Sections .................................................. 9
   4.3 Central Loop .................................................... 11
5 Model Checking ....................................................... 13
6 Experiences with UPPAAL ........................................... 15
   6.1 General Impression ............................................. 15
   6.2 Some Problems ................................................. 15
7 Summary ............................................................. 16
8 Conclusion ............................................................ 16
Bibliography ............................................................ 16
1 Introduction

Tilt-tray sorters are typically used in express parcel systems

Fig. 1: An impression of a simplified express parcel system configuration.

The design of such systems has evolved over the last years. Different physical appearances of conveyors have been developed, and the behaviours of complete systems have been simulated. One of the remaining problems is the uncertainty of designers concerning the optimality of the applied scheduling algorithms. They can verify chosen scheduling approaches, but cannot develop scheduling algorithms in a systematic way.

The sorter under study in this report is the tilt-tray sorter, a conveyor consisting of relatively small segments that can be tilted independently, transverse to the movement direction. The general idea of tilting is that getting packages off the sorter is simple. In many cases a package will cover multiple segments. The limited size of each segment has a positive effect on the throughput capacity of the system. An example of a tilt-tray sorter is shown in Figure 2.

2 Problem Statement

In this research the main question is whether it is possible to apply \textsc{Uppaal} as a means to study scheduling rules in a more systematic way and to achieve either supporting evidence for heuristic scheduling rules or find new, more optimal rules. In this sense, learning the possibilities and limitations of \textsc{Uppaal} were certainly as important as the case study itself. One of the most challenging issues for the design of systems that involve tilt-tray sorters is to determine the suitable combination of the following parameters w.r.t. cost and performance (in Figure 3 this refers to the system part at the bottom):

\footnote{See \url{http://www.vanderlande.com}.}
Figure 2: A tilt-tray sorter where an item covering three segments just gets off.

Figure 3: A schematic overview of the classes of system variables.

- physical sizing of the sorter, physical distances between the inputs (and outputs);
- capacity of the central sorter in packages per hour (the speed in terms of number of segments per second);
- the size of the segments;
- capacities of the inputs (and outputs) (how many packages per hour can be handled per input or output);
- number of inputs (and outputs);
- scheduling rules for putting packages on the sorter.

Most of the output-related parameters are put in between brackets because they mainly depend on the customer situation and not so much on the system limitations. Often, the output capacity of each individual output will be larger than the capacity of the central sorter, which means that it is not a limiting factor.

Next to these parameters, the distribution of package sizes is very relevant. It is easy to imagine a “starvation” situation when a greedy scheduling algorithm (for putting packages
onto the sorter) is combined with the large packages being fed to the last input. When the utilisation of the sorter is high, the large packages will never get onto the central sorter. In Figure 3 this refers to the system usage part at the top. In most simulation studies, such systems are modelled following all design decisions for the abovementioned parameters that have been made before, together with the package size distribution that is taken from a customer case, and the simulation then reveals the resulting system behaviour. That is, static design parameters and system usage are varied in a controlled manner (cf. Figure 3). Variations of these parameters then lead to analysis of a larger design space, but this is difficult to achieve with respect to the scheduling rules. Since scheduling rules are subject to trial-and-error engineering, some basic principles are quite well understood, while other rules are validated through simulation. Thus, the dynamic design parameters are mostly not subject to controlled variations.

3 Approach

The approach taken in this study is different from the regular simulation approaches. The models that have been made all have in common that the static design parameters are still in the model, but the scheduling rules are not. The scheduling is viewed as a dependent parameter, i.e., an output parameter or outcome of the model, not as an independent parameter, i.e., an input parameter. This approach is chosen, because the (heuristic) scheduling rules constitute one of the most fuzzy parts of designing such a system, and more insight c.q. improvement is aimed for. UPPAAL has been used in other case studies in this way [2]. This tool applies timed automata as the formalism to express models, and offers an integrated model checker as well. Moreover, the model can be simulated for random traces as well as traces that are generated by the model checker, which either illustrates the existence of some stated property by a concrete example or, the other way around, provides a counter-example for a property that does not hold. The UPPAAL model only represents the physical behaviour and constraints that are caused by the components used. That is, the actual scheduling rules are not modelled, while speeds, connections, and constraints on timings caused by limits on accelerations and speeds are included in the model.

4 Evolution of Models

Due to several reasons models evolve over time. Almost always, only the last model is presented in a report. Here a number of evolutionary steps are discussed to indicate the versatility in modelling the current problem. In Figure 4 the first idea of a complete model is sketched. The different blocks indicate independent automata, i.e., they progress in parallel and can synchronize on specified, pairwise transitions. On the left-hand side two inputs are modelled, each with a generator (generating the packages) and an induct section, a speed-controllable conveyor that “shoots” the packages onto the central loop. The loop is the middle block. On the right-hand side two outputs are modelled, very much alike the inputs. The global variables of the model define the structure box, which has an identification number, length, source, and destination. A meta box is used to exchange packages between different automata, which is explained later. Further, when packages are created, their properties are generated by global functions. The synchronization between automata

---

3 The structure box is defined to have its four fields in the following way:
   typedef struct { int nr; int[0,MAXLENGTH] length; int[0,INPUTS] source;
   int[0,OUTPUTS] destination; } box;
4 Meta variables are stored in the state vector, but are semantically not considered part of the state. I.e. two states that only differ in meta variables are considered to be equal.
Tilt-tray Sorters modelled with UPPAAL
ESI Report Nr. 2008–10528
Version 1.0

Figure 4: An overview of a complete model for a typical configuration with 2 inputs and 2 outputs. The different blocks indicate independent automata in which schematically the state transitions structure is shown.

is done over channels urgent\textsuperscript{1} chan transferBox[{4},[20]], where the first index is coinciding with the size of the package, and the second index is just a numbering of channels over the complete model.

The model declaration consists of a list of automata that together form the system: \texttt{system G, CI, L, CO, E} (where the letters denote Generator (G), Conveyor Input Section (CI), Central Loop (L), Conveyor Output Section (CO), and Exit (E)). Often this expression contains a typical usage of UPPAAL of the form \texttt{G(const int[0,1] i) = BoxGenerator(i,1)}, which automatically generates two automata \texttt{G(0)} and \texttt{G(1)} when \texttt{G} is specified in the system, which are equal to \texttt{BoxGenerator(0,1)} and \texttt{BoxGenerator(1,1)}. The second parameter of \texttt{BoxGenerator}, which denotes the number of its communication channel, can also be changed through the introduction of an additional array that labels the connections between the automata. In such a way the generation of automata and their connections is very easily specified.

After the first model checks (see Section 5) it appeared that model checking was rather slow. Moreover, the details of the induct sections seemed not too relevant for the problem at hand, because scheduling on the loop is the central problem in the case under study, not the specific control of a particular induct section. In the end, the induct sections were removed altogether, while their delays (before being able to accept a new package) were included in the main loop. Figure 5 depicts the complete model in this case.

The system models at different stages of evolution sometimes are equivalent, sometimes they are not. The equivalence is "achieved" by human reasoning about the system behaviour, and subsequent modeling by hand. Sometimes, one would want UPPAAL to be as smart to propose the model changes by itself: a good example is shown in Section 4.3 about the model evolution of the central loop. The third version of the model explicitly picks one gauge of a certain symmetry in earlier models which improves the performance (of model checking) drastically, but does not alter the meaningful aspects of the model.

4.1 Package Generation and Exits

Generation of packages and exits of the system are modelled as trivially as possible as can be seen in Figure 6.

The generation automaton checks for the total number of required boxes, i.e., box\_nr <\textsuperscript{1}\textsuperscript{1}Urgent channels enforce transitions to be taken without any time delay.
4.2 Induct Sections

Induct sections are used to “shoot” packages at the right location onto the central conveyor loop. This is done by stopping the package, until the correct time has been reached, after which it is accelerated to some adjustable speed and arrives at the central belt at the correct position in time. The acceleration can be done by several independent segments of the input conveyor belt, while every segment may not occupy more than one package at any given time. These segments together form an induct section. For our purposes, i.e. finding possible schedules on the central loop, a sufficient model enforces that packages can only be delivered at a certain frequency, not higher but possibly lower.

In Figure 7 two versions of the induct section model are shown. The states (also called locations) are indicated by filled circles and can be given recognizable names like IDLE, READY, ... These states also can have so-called invariants as $x \leq T$ or $x \leq \text{delay}$ in which $x$ denotes a local clock. These invariants always have to hold and act as a kind of dispersers (inverse guards) on the outgoing transitions: $x \leq T$ means that somewhere before $x$ gets a value
larger than $T$ the location has to be left.

To explain the left-hand side model in somewhat more detail, the states and transitions are listed below together with their explanation:

- **IDLE** is the state that no package is on the section, but it is not ready to accept a new package. The state has to be left for clock value $x$ smaller or equal than $T$, a variable that is calculated in the last transition (see below) and initially is set to zero.

- **IDLE** to **READY** transition is taken at clock $x$ value larger or equal than $T$, which in combination with the previous invariant means that the transition is taken at exactly $x = T$.

- **READY** is the state where the section is empty and can receive a new package.

- **READY** to **LOADING** transition receives an actual package of size $s + 1$ over the input channel $\text{transferBox}[s][cI]$. The parameter $s$ is selected from a range of integers $[0,3]$. Receiving is indicated by the question mark $?$. The update assignments involve resetting the local clock ($x = 0$), picking up the package in the local variable $\text{box bb}$ by copying it from the global meta box $b$, and setting the local variable for size.

- **LOADING** is the state in which the front edge of the package is on the section. This takes exactly delay time (cf. the first transition).

- **UNLOADING** is the state in which the package is ready to leave the section, i.e., the front edge is ready to go to the central loop. This state cannot be achieved within delay time units coming from READY, but can last longer. This is the case when the speed of the induct section is reduced.

- **UNLOADING** to **IDLE** is the transition by which the package is sent to the next automaton. This is done over the output channel $\text{transferBox}[size][cO]$. Sending is indicated by the exclamation mark !. The update assignments involve setting the global meta box $b$ variable (in order to transfer the correct package characteristics) and setting $T$, the time that it takes to remove the package completely from the section. This time $T$ is dependent on the package characteristics through the function $\text{packet\_delay}$.

The communication of variable values cannot be done directly within UPPAAL, as the communication channels do not transfer data. Nevertheless, through global variables it is possible to exchange variable values. The update instruction of the sender is always done before the
update instruction of the receiver, and therefore the mechanism as shown on the left in Figure 8 works, i.e. after synchronization \( \ell_2 \) has the same value as \( \ell_1 \) through the exchange over global variable \( g \).

The packages that are used vary in size between 1 and 4 units of length (tray width of the tilt-tray sorter). The transfer of packages between automatons is done over bundles of channels, which can be observed in the selection of the parameter \( s \) (which can be interpreted as size-1) and the synchronization channels \( \text{transferBox}[s][cI] \) (for the input) and \( \text{transferBox}[s][cO] \) (for the output). The reason for this is that certain transitions can be enabled for certain sizes but not all sizes, which is visible in the dependence of the guard \( \text{guard}(s) \) on \( s \) (see middle picture of Figure 8).

The second model of the induct section (see Figure 7, right-hand side) removes a part of the unnecessary complexity of the first model. It only has two states, and involves no calculation of a size-dependent delay anymore. This improved the performance of the model (in terms of model checking) significantly.

### 4.3 Central Loop

The first model of the central loop is shown in Figure 9. The occupation of the trays on the central loop by packages is modelled through an internal array called \( \text{contents[]} \). The functions \( \text{setContents()} \) and \( \text{getContents()} \) interact with this internal array in

---

**Figure 8:** Some basic patterns in UPPAAL. On the left-hand side variable exchange between two automatons is shown. In the middle the use of multiple channels is shown. On the right-hand side time progression in combination with zero-duration actions is shown.

**Figure 9:** One of the first versions of the central loop. On the left-hand side the input transitions are shown, on the right-hand side the output transitions, while on the top movement of the belt is modelled. The input and output transitions have to take place at \( x = 0 \), while the belt movement is an internal update action \( \text{shiftWindow()} \) at \( x = 1 \). Note that in such a way the central loop can handle 4 package movements concurrently within a single automaton.
a straightforward way. The locations of the inputs and outputs as well as the position of the central loop is represented through the offset with which the `contents` array is accessed. The position of the loop is shifted with the function `shiftWindow()`, which is in the update part of the transition that takes place at local clock $x = 1$. So every unit of time the loop sorter is shifted one position. This pattern is also shown in Figure 8 on the right-hand side. The `space()` function serves as a guard on the input transition: if there is no space for a package, it cannot be transferred. This is also where the different channels for different sizes come in: dependent on size the transition should become enabled or not. In this way this is achieved elegantly. This pattern is also shown in the centre of Figure 8. Here it is clear that the guard `guard(s)` blocks/enables the transitions on basis of the package sizes.

In a next version of the central loop, the limitation on the speeds of the inputs is included in the loop model. In this way the need for separate induct sections completely disappears. This is shown in Figure 10 in which the inputs are not only guarded by `space()` but also by clock constraints of the form $x_i >= d_i$. This model change was done to improve the performance of model checking once more. The model has identical behaviour, although this was not explicitly verified.

In the process of model checking (see next section) it appears that UPPAAL sometimes needs much time to check certain properties. If this is combined with the intention to scale the model to more inputs (8 instead of 2 was tried), the model soon becomes useless in terms of necessary calculation times. One approach to tackle this problem is shown in Figure 11 in which the order in which transitions can take place is fixed. Through this modification the number of possibilities that the program has to check reduces from $9!$ to $2^9$ for each complete cycle in the central loop (i.e., until the loop is shifted one position further). This is a factor 700 of difference.

A disadvantage of the last approach is the “engineering” of the model that takes place, which only serves tractability for UPPAAL. Furthermore, it becomes more difficult to include heuristics in the model, which can be applied in UPPAAL to enable faster search strategies in state space. These heuristics are comparable to cost functions that guide UPPAAL in determining which option has priority over other options.

Note that the loop model in Figure 11 has a completely different visual appearance, and that the loop in this graph has nothing to do with the physical loop it represents. Further, the model is larger in terms of locations and transitions, but nevertheless faster. For some this might be somewhat counterintuitive.
5 Model Checking

A model checker is built into UPPAAL. It yields the possibility to verify properties of the model that one has built. One of the most common things to check is the absence of deadlock:

\[ A[] \neg \text{deadlock} \]

In the models of the tilt-tray sorter this is a quite trivial check, because the central loop always can proceed.

Next, a performance property can be tested. This is central to the approach as sketched in the introduction. What we are looking for are concrete schedules for which a certain performance is achieved. One can do this by checking the following formula:

\[ E<> (\forall i : \text{theexitnr} . E(i).\text{boxes}=\text{NBOXES/OUTPUTS}) \text{ and } \text{now} < \text{TIME\_ESTIMATE()} + 5 \]

in which \text{now} is a globally declared clock.

In a simpler case of only one output (the capacity of which is larger than the central loop, so this simplification is not a real limitation) this reduces to

\[ E<> E(0).\text{boxes}=\text{NBOXES} \text{ and } \text{now} < \text{TIME\_ESTIMATE()} + 5 \]

Figure 11: A third version of the central loop. In this case the model enforces a certain order in checking whether input transitions are possible or not. For every next step either a transition occurs, or nothing is done. The last transition on the right is the output transition.
The number 5 in these formulae is subject to manual tuning. The property that is expressed is the following: there is a trace of the model that leads to the state in which the exit automaton $E(0)$ has received $N\text{BOXES}$ and the global clock $\text{now}$ is less than the function $\text{TIME\_ESTIMATE()}$ increased by 5. The function depends on the distribution of packages that is fed to the system – it is equivalent to the time that all packages need on the sorter with full occupation.

It is not difficult to find the minimum number (here 5) for which the formula holds. Subsequently, UPPAAL can be asked to generate a trace that indicates that the formula holds. Such a trace can be exported from the system, transformed into human-readable form (by a tool called libutap), and filtered (e.g. by gawk) to usable input for a plot program such as gnuplot.

An example output is shown in Figure 12. The graph shows a trace that was generated by UPPAAL that is not optimal or fastest, although this cannot directly be inferred by the graph only: also the capacities of the inputs could be so low that higher utilisation of the sorter is not achievable. The reason for not showing an optimal graph is directly related to an improvement point for UPPAAL: for smaller values of the end-time to be verified, UPPAAL gave an error message. This means that either the model still contains deficiencies or there is an error in UPPAAL itself. This issue has not been resolved up to now.
6 Experiences with UPPAAL

6.1 General Impression

The general impression of UPPAAL is very positive: it has a professional, quite mature appearance and it hardly crashes. The graphical capabilities make working with it enjoyable and fast, although the tool in itself needs a little learning curve which is fastest through learning-by-example. It is the combination of modelling environment, trace simulator and model checker that makes UPPAAL so valuable.

On the critical side of things: what is being called large scale models by some people, other people consider rather small and unrealistic in industrial environments. In that sense, UPPAAL still has it clear limits in applicability over different domains. The models tend to become quite abstract for some domains: in the problem under study the central loop representation has become difficult to explain in only a few sentences and does not materialize quickly with people that are new to the formalism.

6.2 Some Problems

Some limits of UPPAAL have been encountered frequently in the course of working on this problem. A concise list of issues that are still open is useful either for follow-up actions in the UPPAAL community or for consideration of someone that is going to apply UPPAAL on these type of problems.

1. Trace to error-in-model not implemented.

Debugging facilities are important, irrespective the level at which one works. For code generation it is indispensable, as it is for model building. At too many instances it was unclear where the problem could be; especially when the error only becomes apparent in the model checker, the absence of the trace to the error condition is really a nuisance. The syntax checker at the modeling stage itself is quite adequate.

2. Difficult to build your own mental model of the state space.

It is cumbersome to estimate the effects of changes that one makes in the model. In terms of locations and transitions this is still doable, but the addition of variables and meta-variables really make it inextricable. Some visualization could help, maybe alike state space visualization as described in [1], alternatively explicit numbers that indicate its size could help. The options that the user has in doing model checking (breadth first, depth first, etc.) are not transparent. Actually, as a user you would want the programme to choose the best option for you.

3. Visualization of traces is limited to step-through and sequence diagram types.

For any model that is somewhat larger than a few processes, sequence diagrams and stepping through is appropriate for solving model errors, but not for studying the modeled system in more detail. An example of visualization is shown in Figure 12. A seamless I/O to other tools could also be an option to alleviate this issue.

4. Trace variables not correctly exported.

This is a strange topic. Although all the information is present in UPPAAL, the state variable information is not exported in a trustworthy manner. Figure 12 could only be made for a subset of the models, because for others the exported values just did not match the values that were visible within UPPAAL itself.

Moreover, being able to chose what information to export can be fruitful, as this constitutes also a solution for the next item.

5. Trace of UPPAAL CORA cannot be exported, nor can it be read in UPPAAL.

Larger models become difficult to verify because of the calculation time needed. One possible solution for this is to load the model into UPPAAL CORA, in which one can add...
heuristics to the search strategy. This can help to traverse the state space tremendously faster, although this was not worked out in detail in this research because of the reason below.

A problem with the CORA variant is, that the exported traces could not be handled by libutap, nor imported into the “normal” variant (in the latter case, a work-around would be at hand).

6. Overview over your model.

It is difficult to keep overview over your model. The automata are clear as templates, and their instantiations are shown in the simulation window, but scaling of them is not as easy as should be. Declarations are divided over different sections “declarations” and “system declarations”, which has a somewhat logical division (sequentiality in definitions) but is not really convincing.

7 Summary

The goal of this research, that is to improve on the heuristic scheduling rules for tilt-tray sorters, was not reached. Actually, the UPPAAL limitations and problems were the main cause. It is not guaranteed, that this approach would have been successful in the absence of these problems and limitations, as deducing sound heuristics from generated schedules could be quite difficult.

A number of research issues are still on the table. Applying cost functions in the model and use this in UPPAAL CORA is one of them, doing the appropriate design-of-experiments is another one. Dealing with the last two remarks in the previous Subsection is a clear supposition for starting this.

8 Conclusion

UPPAAL is a powerful tool for studying, understanding, and proving properties of system dynamical properties. It is very good at the things it was designed for (mainly verification and behavioural simulation), it has currently some deficiencies in adjacent fields of attention. In my experience modeling in UPPAAL is a mix of trying to understand the essence of the problem you want to solve and applying the strength of the tool best. It is difficult to keep this in line with the relevance of the problem in a real world setting. Therefore, my recommendation would be to apply UPPAAL only in those cases where the problem statement is very crisp.

Bibliography
