Design Space Exploration with Generated Timed Automata

Radboud University Nijmegen
Matser Thesis Computer Science
August 18, 2010

Author:	F.H.M. (Fred) Houben
Thesis number:	636
Supervisor:	prof. dr. F.W. (Frits) Vaandrager
External Supervisor:	dr. L.J.A.M. (Lou) Somers
Second Corrector:	prof. dr. J.J.M. (Jozef) Hooman

Preface

This master thesis marks the end of my study Computer Science at the Radboud University Nijmegen. It is the result of ten months of research carried out at Océ Technologies and the Radboud University. During this time I conduced my research within the Octopus project, which is a cooperative research effort involving Océ, the Embedded Systems Institute (ESI), and several academic research groups. I would like to thank Océ for giving me the opportunity to conduct my research in a stimulating and professional environment.

It would not have been possible to write this thesis without the support of a number of people. I would like to thank my Océ supervisor, Lou Somers, for helping me understand the technical difficulties involved in designing printer/copiers and for his feedback on my thesis. I would also like to thank my direct colleagues at Océ and the people involved in the LOA1 Octopus project for their ideas, support and the positive work environment they created.

At the Radboud University I want to thank my supervisor, Frits Vaandrager, for his help with the formal definitions and his support and feedback during the whole project. I also want to thank Georgeta Igna for the experiments she performed that enabled a comparison between her models and the models presented in this thesis.

Last but not least, I want to thank my family for their support. Special thanks to my mother for correcting my English and my girlfriend for supporting me during my studies and helping me through the difficult times.

Fred Houben
Tegelen, 21 juli, 2010

Design Space Exploration with Generated Timed Automata

Fred Houben

Institute for Computing and Information Sciences
Radboud University Nijmegen, The Netherlands

August 18, 2010

Abstract

Modern high-tech embedded systems are constructed from a complex mix of hardware and software components. An important issue during the development of such complex embedded systems is getting to grips with the extremely large number of design options that are available. This is often referred to as the design space of a system under development. To support developers in their exploration of the design space we use a high-level specification that captures possible design options. This high-level specification is then automatically translated to timed automata that are used for design space exploration, i.e. reachability properties are formulated and the state space is exhaustively searched to check whether these properties hold. The results are then fed back to the developers of the system, e.g. with a Gantt chart that visualises the exact execution times of the various system tasks. Based on this information the developers can improve their design. We have applied the implementation of this translation to an Océ case study that focuses on the digital data path of a printer/copier. The results were compared to manually constructed models created for the same case study. Globally the results of this thesis are: (i) a high-level representation that is geared towards designers of embedded systems; (ii) a translation from this high-level specification to timed automata; (iii) an implementation of this translation that can be used to automatically generate timed automata; (iv) a comparison between generated and manually constructed timed automata.

1 Introduction

Many of today's high-tech embedded systems are built out of a complex mix of heterogeneous hardware and software components. These components form a computational platform that processes a combination of environment events, with real-time requirements, and data intensive tasks. Furthermore, these platforms often need to support concurrent or distributed computations. Some examples of such systems are mobile phones, printers/copiers, wafer steppers, and medical scanners. Due to the complexity of these embedded systems the time and resources needed for development are increasing quickly. This increases the challenge of developing embedded systems that are cost effective and at the same time satisfy customer expectations. An important issue during development is to cope with the extremely large number of design options that are available. This is often referred to as the design space of a system under development. Besides functional requirements, quantitative properties like timeliness, resource usage, and energy usage are also important. The relation between the design options, e.g. number of CPUs, memory setup, and scheduling policies on one hand, and the quantitative properties on the other hand are often difficult to establish.

Océ Technologies, the Embedded Systems Institute (ESI), and several academic research groups are involved in a cooperative research effort called the *Octopus* project. The research focuses on the development of new techniques that can be used during the development of embedded systems. One of the research topics is the development of a framework that supports developers in their

exploration of the design space. This framework provides a model-based approach that helps the developers to create a formal model of their system. This process is automated as much as possible and does not require the end-users to be experts in formal methods. There does not exist a single modelling approach or analyses tool that can answer all questions, i.e. each approach tends to have its advantages and disadvantages. The Octopus framework combines the power of several formal method tools into one design space exploration framework, giving the developers the possibility to choose the most appropriate tool for the problem at hand.

The framework is based on the Y-chart [24, 25] methodology depicted in Figure 1. The central idea of the Y-chart is that the development of embedded systems requires the co-development of an *application*, a *platform*, and a *mapping* of the application onto the platform. All three aspects, individually or combined, have an impact on the performance of the system under design. An advantage of this clear separation is that it allows independent evaluation of one or two aspects while the other aspects remain fixed. A good example can be found in the domain of digital printer/copiers. In this domain the emphasis is on the exploration of platform and mapping. The set of applications, for example, copying, scanning, and printing is relatively stable and well known. In other domains the emphasis may be on application and mapping because the platform is fixed. The application, platform, and mapping are used to generate models for one or more tools. The actual *analyses* of the generated models are done by tools that return *diagnostic* information. This information is then used to improve the application, platform, and/or mapping (indicated by the light bulbs).

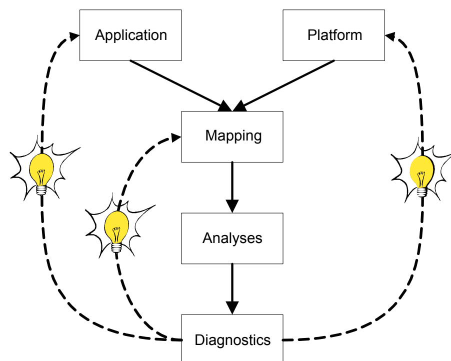


Figure 1: The Y-chart

In this research we first capture the three Y-chart aspects application, mapping, and platform. This is done with the Design Space Exploration Intermediate Representation (DSEIR) that represents a specific design option from which models are generated for analyses. The new concept of *Parameterized Partial Order* (PPO) [15] is used to describe the application aspect. These PPOs are used for specifying task graphs with repetitive behavior. The main advantage of PPOs is that they are simple in use yet expressive enough for practical applications. Another advantage is that they have a well defined formal definition. The next step is to capture the platform aspect. For this we use a generic description of resources that describes for each resource the capacity and the pace at which it can process data. The last aspect is the mapping that describes how an application uses the resource of the platform. The advantage of DSEIR is that several models can be generated from a single specification, which reduces the chance of inconsistencies between models. Another advantage is that DSEIR is more geared towards designers of embedded systems, i.e. it provides a high-level description of a system. By adding domain specific languages on top of DSEIR we can create a representation dedicated to a particular problem domain. These domain specific languages are then translated to DSEIR, which in turn is translated to the various models. However, in this research we consider DSEIR as the top-level language and we regard the addition of domain specific languages as future work.

Given a DSEIR representation that captures the three Y-chart aspects we define a translation to UPPAAL. This is a tool for modeling and verification of timed systems modelled as networks of timed automata [5, 6]. In previous work timed automata have been successfully used for optimal planning and scheduling problems [1, 2, 16, 18, 20], and performance analyses of real-time distributed systems [19, 28]. With this translation UPPAAL models can be generated for the purpose of design space exploration. Figure 2 shows an implementation of the framework for two different modelling approaches and corresponding analyses tools. Here files are used to denote input and output and rectangles denote tools that transform input to output. The first (top) toolchain generates coloured petri net [21] models and uses CPN Tools [22] for simulation. The second toolchain generates UPPAAL models and uses the UPPAAL model checker to explore the state space and generate a trace file. From this trace file a Gantt chart is generated that helps the developers in improving their design. This last toolchain is marked grey and describes the files and tools used in this research.

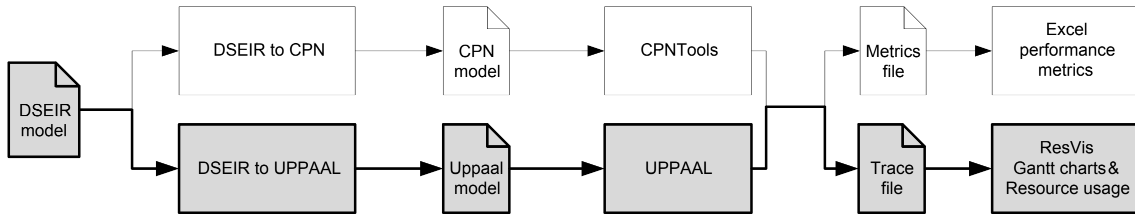


Figure 2: Octopus framework realisation

Research questions. In this master thesis the following research questions are investigated:

- What is the formal definition of task-level parameterized partial orders?
- How to translate task-level parameterized partial orders to UPPAAL?
- How to extend the translation with resources?
- How do the generated models compare to manually constructed models?

Case study. The industrial case study used in this research is provided by Océ technologies, a manufacturer of digital document printer/copiers. These printer/copiers are embedded systems with soft real-time requirements that process all kind of paper-based and data only documents. The focus of the case study is on the digital *data path* that transforms input to output data. This data path consists of various resources like FPGAs, CPUs, memory, and busses and forms the platform on which computations are performed. This platform executes various applications like copying, scanning, and printing. The main challenge is to compute efficient schedules for applications that minimise execution time on a platform with limited resource. Other important aspects of digital document printer/copiers, that do not fall within the scope of the case study, are the paper flow and the physical printing process.

Related work. In previous work, within the Octopus project, several modeling approaches and analyses tools have been applied to the case study described above. In [20], the case study is modelled and analysed using three different approaches: time automata, coloured petri nets, and synchronous data flow. These approaches are compared in term of found solutions, ease of model construction, and analyses efficiency. A more elaborate description of the coloured petri net model is given in [23]. In [4] the time automata model is modified to solve scheduling problems with uncertainty, i.e in reality the arrival times of new print jobs are typically unknown. Timed game automata are used to model uncertainty in the arrival times of new jobs. All the research mentioned above use manually constructed models for the case study. This increases the chance

of inconsistencies occurring between the various models. Another drawback is that experts are needed to create and maintain these models. Even worse, a different expert my be needed for each modeling technique. In this research we address these problems by generating timed automata from a single, easy to use, specification. A key role in this specification is the used of PPOs for specifying task graphs with repetitive behavior. In [17] another approach for specifying compact task graphs with repetitive behavior is described. Here a subclass of UML activity diagrams are used to specify the causal dependencies between tasks. The main disadvantage of this approach is the complexity of the activity diagrams that makes them less intuitive than PPOs. Other non-repetitive approaches for specifying task graphs are given in [2, 27, 30].

Running example. Throughout this document a simple running example is used to illustrate various points. This example consists of two tasks a and b. The idea is that the output of a is the input of b. It describes a common situation in which task a writes some data into a buffer which is then read by task b. Depending on the buffer size b could wait for a to fill the buffer and finish, after which b starts reading the buffer. Instead of waiting for a to finish b can also start processing chunks of the output from a. This streaming behavior is common in, for example, video encoding/decoding, graphic rendering pipelines, and the processing of documents in printer/copiers. In the domain of digital document printer/copiers the scanning and resampling of a document represents such a case. Here a represents the *scan* task that scans a document line by line and writes these lines into the buffer. Task b represents the *resample* task that can start when the buffer contains enough lines.

Organisation. This master thesis is organised as follows. In section 2 we introduce the concept of parameterized partial orders that are used for specifying applications. Section 3 gives a short overview of the Uppaal modelling language and tools. Based on the previous two sections a translation from parameterized partial orders to Uppaal timed automata is given in section 4. With this translation in hand, section 5 describes the implementation. The next step is to extend the translation with a platform and mapping that describe how applications are executed. This is described in section 6. In section 7 we verify the behavior of the generated models by comparing them with manually constructed models. Besides behavior we also compare the performance during model-checking. Finally, section 8 gives the conclusions and directions for future work.

2 Partial Order

The application aspect of the Y-chart describes a number of tasks that perform some functionality of a system. To represent the tasks of an application we use the concept of Partial Orders (PO). A PO is a binary relation that describes the ordering of elements in a set. It is called a *partial* order to reflect the fact that not every pair of elements in the set needs to be related. With a PO one can describe the causal relation between tasks. In other words, one can describe that some task have to be executed in sequence while others can be executes concurrently. For example, in the domain of digital document printers, a PO can be used to specify the causal relations between tasks involved in realising applications like copying, scanning, and printing. In fact, applications in many different domains can be modelled as a PO. This section first defines the standard model of event-level POs. The next step is to extend the definition to parameterized POs that can be used to compactly represent POs. The last step is to define taks-level parameterized POs that are more intuitive and enable a more efficient translation to Uppaal (see section 4).

2.1 Event-level Partial Order

The standard model of Event-level Partial Orders (EPO), as described by e.g. [26, 33], can be used to describe applications. Such an EPO describes precedence relations between events. An event identifies an instantaneous action that marks state change. Each event is unique and can occur only once. The causal ordering between events is defined by the *precedence* relations that

describe in what order events should occur but not the precise time at which events occur. We abstract from the precise time because this depends on who or what executes the application. In a computer system the precise time is determined by the specifications of resources like memory, CPU, and buses. Another example are the applications of a manufacturing plant that are typically executed by a mix of human labour and machinery. This means that there is a clear separation between applications, described as an PO, and the resources on which they are executed. At a later stage resources and clocks will be added to give a more precise notion of time (see section 6).

Events can be used to specify tasks by pairing *start* and *end* events. The duration of a task is defined by the time that elapses between the occurrence of the two events. A restriction is that the start event must precede the end event of a task.

An EPO is a pair (E, P) where E is the set of events and $P \subseteq (E \times E)$ is a binary precedence relation on the set E. Intuitively a precedence pair (a, b) describes that event a must precede event b. If we want an EPO (E, P) to describe a meaningful ordering of events we need to impose three restrictions on the binary relation P.

- P is transitive, i.e. $\forall a, b, c \in E : aPb \land bPc \rightarrow aPc$
  Intuitively this says that whenever a precedes b and b precedes c then also a precedes c.
- P is irreflexive, i.e. $\forall e \in E : \neg ePe$
  The intuition behind this is that no event can precede itself.
- P is asymmetric (not symmetric), i.e. $\forall a, b \in E : aPb \rightarrow \neg bPa$
  This restriction avoids cycles. In other words, an event that precedes a sequence of one or more events cannot depend on that sequence.

An EPO can be represented by a directed acyclic graph, where the nodes denote events and directed edges denote precedence relations between events. The graph in figure 3 shows an EPO (E, P) with $E = \{as, ae, bs, be\}$ and $P = \{(as, ae), (bs, be), (as, bs), (ae, be), (as, be)\}$. Note that the pair $(as, be) \in P$ is not drawn in figure 3 because it follows out of the transitive property. The graph is drawn with the fewest possible edges such that it is still represents the same reachability. This is also known as a Hasse diagram which is visualisation of the transitive reduction.

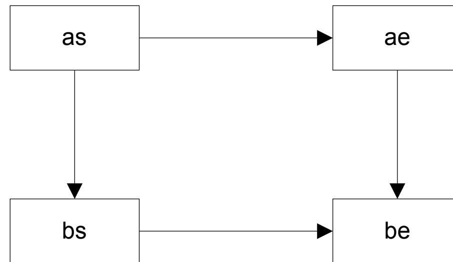


Figure 3: Example of an EPO

The EPO in figure 3 describes the ordering of two tasks a and b. Each task is specified by a start and end event and a precedence relation that ensures that the start event precedes the end event. So the nodes as and ae together with the precedence edge (as, ae) represent task a. Task b follows the same scheme. The idea is that the output of a is the input of b. Instead of waiting for a to finish b can start processing chunks of the output from a after a has started (precedence (as, bs)). As b depends on the output of a it can only finish when a has finished (precedence (ae, be)). This describes a common situation in which task a writes some data into a buffer which is then read by task b. This example will be used as a *running example* throughout the document to illustrate various points.

2.2 Event-level Parameterized Partial Order

Applications often contain patterns that repeat in a predictable manner. Specifying such repetitions in an EPO, as described in the previous section, results in very large POs. Another drawback is that one does not have a finite representation for an arbitrary number of repetitions, i.e. there is no finite representation for an infinite EPO. In order to facilitate large or infinite EPOs we introduce Event-level Parameterized Partial Orders (EPPO) [15], which can be used to represent recurring events and precedence rules in a compact manner.

Event classes are used to represent events that are repeated several times. Each event class is parameterized with one or more instantiation variables and a specific instance is defined by a valuation of these variables. To simplify the notation, variable names can be reused in different event classes, e.g. both event classes A and B can have an instantiation variable x. If there are infinite valuations for the instantiation variables we end-up with an infinite EPO, i.e. there are infinite instances of event classes. To restrict the number of possible instances, each event class has a range that defines the lower and upper bounds. The precedence relations between classes are extended with boolean expressions that describe when a precedence relation between two event class instances exists. These expressions are build over the instantiation variables of the two classes that have a precedence relation. Because variables names can be reused we need a way to distinguish between the source and target class variables. This is done by decorating all shared variable names of the target class with a prime. We will now present the formal definition of EPPOs.

Let VAR be the universe of typed variables. A valuation of a set $A \subseteq VAR$ is a function that maps each variable in A to an element of its domain. We write $V(A)$ for the set of valuations of variables in A. Let $UVAR$ be the set of undecorated variables and $DVAR$ the set of decorated variables defines as $DVAR = \{v' \mid v \in UVAR\}$. Then the set of all variables is $VAR = UVAR \cup DVAR$. Given a set $A \subseteq UVAR$ we write A' for the set $\{a' \mid a \in A\}$. The set *BoolExp* of boolean expressions is built over function symbols, predicate symbols, constants, and variables. We write *BoolExp*(A) for the set of boolean expressions built over the variables in set A.

An EPPO is a tuple (E, P, M, R, C) where

- E is a finite set of events classes.
- $P \subseteq (E \times E)$ is the set of precedence relations between event classes.
- $M : E \to \mathcal{F}(UVAR)$ associates to each event class e a finite set of instantiation variables. Here $\mathcal{F}(UVAR)$ is the set of all finite subsets of $UVAR$. We write $V(e)$ as a shorthand for $V(M(e))$.
- $R : E \to$ *BoolExp* gives the allowed valuation range for the instantiation variables of an event class. A range is built over the instantiation variables of the associated event class. Therefore we require, for all $e \in E$, $R(e) \in$ *BoolExp*$(M(e))$.
- $C : P \to$ *BoolExp* gives the condition under which a precedence relation between two event class instances exists. Elements of $UVAR$ denote the variables of the *source* event class and elements of $DVAR$ denote the variables of the *target* event class. We require, $C(s,t) \in$ *BoolExp*$(M(s) \cup M(t)')$.

Figure 4 shows the EPPO for the running example that is defined by:

$$E = \{as, ae, bs, be\}$$

$$P = \{(as, ae), (bs, be), (as, bs), (ae, be), (be, as))\}$$

$$M(as) = M(ae) = M(bs) = M(be) = \{p\}$$

$$R(as) = R(ae) = R(bs) = R(be) = 1 \leq p \leq 2$$

$$C((as, ae)) = C((bs, be)) = C((as, bs)) = C((ae, be)) \equiv p' = p,\ C((be, as)) \equiv p' = p + 1$$

1 ≤ p ≤ 2

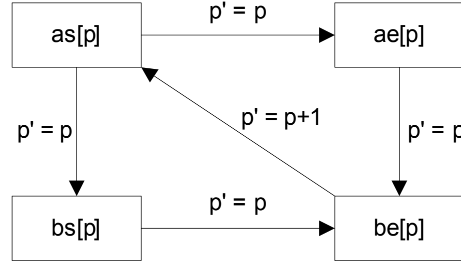


Figure 4: Example of an EPPO

The example describes the situation in which task a writes some data into a buffer which is then read by task b. The event classes are visualised as $as[p]$, $ae[p]$, $bs[p]$, and $be[p]$ where p is an instantiation variable with a different scope for each class. Directly above the event classes is the valuation range of the instantiation variables. If every range contains the same sub condition this sub condition is shown as a global condition outside the graph. This is useful when many event classes have the same sub condition, as in figure 4, because we only have to write it once. Each precedence relation has a condition that describes whether a precedence relation between two event instances exists.

EPPOs add no expressive power compared to EPOs as they are nothing more that a compact representation of EPOs. An EPPO can be *unfolded* to an EPO by assigning all values, within the range, to the instantiation variables. The resulting event instances are ordered with the precedence relations. If the condition evaluates to true then a precedence relation exists between two event instances. Figure 5 shows the EPO after unfolding the EPPO in figure 4.

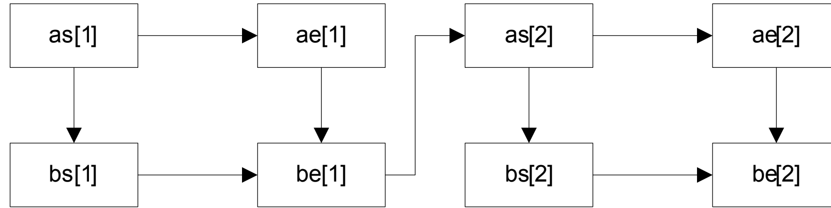


Figure 5: Example of an unfolded EPPO

We will now formally define the unfolding of an EPPO. We use the notation $v \models t$ to denote that boolean expression t is a consequence of valuation v. In other words, $v \models t$ is true if t evaluates to *true* for valuation v. Given an EPPO (E, P, M, R, C) we defined the unfolding to an EPO $(\bar{E}, \bar{P})$ as:

$\bar{E} = \{(e, v) \mid e \in E \wedge v \in V(e) \wedge v \models R(e)\}$

$\widetilde{P} = \{((e, v), (f, w)) \mid (e, f) \in P \wedge v, w' \models C(e, f)\}$
here w' is the valuation of $V(f)'$ given by $w'(x') = w(x)$, for $x \in V(f)$

$\bar{P}$ = the transitive closure of $\widetilde{P}$

Note that after constructing the binary precedence relation we take its transitive closure. This ensures that $\bar{P}$ satisfies the transitive property. In other words, if the relation is already transitive it will not change, otherwise precedences will be added that denote the transitive properties between events. A EPPO is said to be *consistent* iff its unfolding yields an EPO.

2.3 Task-level Parameterized Partial Order

In the previous section we described tasks in terms of events. Events describe instantaneous actions at a certain point in time. However, engineers typically specify system activities in terms of tasks and durations because these are considered more intuitive than events. Task based descriptions explicitly associate tasks with durations, whereas event implicitly describe durations. As we will see, another advantage of lifting the representation to the task level is that it enables a more efficient translation to UPPAAL. In a Task-level Parameterized Partial Orders (TPPO) the tasks are parameterized instead of the events. Each task has a start and end event on which the precedence relations are defined.

A TPPO is a tuple (T, E, P, M, R, C)

- T is a finite set of task classes.
- E is a finite set of events. We assume two functions $start : T \to E$ and $end : T \to E$ that maps each task to its corresponding start and end event. The function $task : E \to T$ maps each event to its corresponding task. For each task t we require, $task(start(t)) = t$ and $task(end(t)) = t$. Furthermore, we require that each event is associated with a task, i.e. $\forall e \in E, \exists t \in T : e = start(t) \vee e = end(t)$.
- $P \subseteq (E \times E)$ is the set of precedence relations between events. To ensures that the start of a task precedes its end we require that, for each $t \in T$, $(start(t), end(t)) \in P$.
- $M : T \to \mathcal{F}(UVAR)$ associates to each task class t a finite set of instantiation variables. We write $V(t)$ as a shorthand for $V(M(t))$.
- $R : T \to BoolExp$ gives the allowed valuation range for the instantiation variables of a task class. A range is built over the instantiation variables of the associated task class. Therefore we require, for all $t \in T$, $R(t) \in BoolExp(M(t))$.
- $C : P \to BoolExp$ gives the condition under which a precedence relation between two events exists. We require, $C(s, t) \in BoolExp(M(task(s)) \cup M(task(t))')$.

Figure 6 shows the TPPO for the running example that is defined by:

$$T = \{a, b\}$$
$$E = \{as, ae, bs, be\}$$
$$P = \{(as, ae), (bs, be), (as, bs), (ae, be), (be, as)\}$$
$$M(a) = M(b) = p$$
$$R(a) = R(b) = 1 \leq p \leq 2$$
$$C((as, bs)) = C((ae, be)) \equiv p' = p,\ C((be, as)) \equiv p' = p + 1$$

Unfolding a TPPO is similar to unfolding an EPPO, i.e. a TPPO can be unfolded by assigning all the values, within the range, to the instantiation variables and ordering the obtained tasks instances with the precedence relations. Figure 7 shows the unfolding of the TPPO in figure 6. A TPPO is said to be *consistent* iff it can be converted to a *consistent* EPPO. Given a TPPO (T, E, P, M, R, C) we defined the conversion to an EPPO $(\bar{E}, \bar{P}, \bar{M}, \bar{R}, \bar{C})$ as:

$$\bar{E} = E$$
$$\bar{P} = P$$
$$\bar{M} = (\bar{E}, \mathcal{F}(UVAR),\ \{(start(t), M(t)) \mid t \in T\} \cup \{(end(t), M(t)) \mid t \in T\})$$
$$\bar{R} = (\bar{E}, BoolExp,\ \{(start(t), R(t)) \mid t \in T\} \cup \{(end(t), R(t)) \mid t \in T\})$$
$$\bar{C} = (\bar{P}, BoolExp,\ \{((start(t), end(t)), \forall v \in M(t) : v' = v) \mid t \in T\} \cup \{(p, C(p)) \mid p \in P\})$$

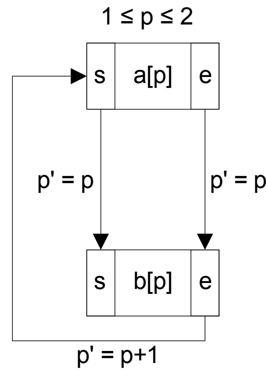


Figure 6: Example of a TPPO

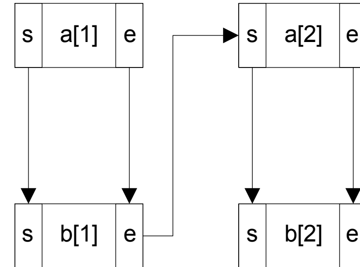


Figure 7: Example of an unfolded TPPO

3 UPPAAL

Uppaal is an integrated tool environment for modeling, simulation, and verification of timed systems. It is designed to exhaustively explore the state space of a modelled system. It has been successfully applied to numerous industrial case studies [9]. The environment consists of three main parts: a modelling language, a simulator, and a model checker. A short overview is given for each of these three parts. In later sections the reader will be provided with additional information when needed. For more in depth information see [9] for a tutorial and [11] for the underlying theory, semantics, and algorithms.

The modelling language is based on timed automata [6, 11]. Such a timed automaton is a finite-state machine extended with continues clock variables. All clocks increase synchronously with the same rate. Besides clock variables, the automata in Uppaal are extended with bounded discrete variables. An automaton is composed out of a finite number of locations and edges between these locations. Edges are decorated with *guard*, *synchronisation*, and *update* labels. An automaton may fire an edge separately when the guard is satisfied. Edges from different automata can also be fired synchronously when the guards and synchronization labels of all involved edges are satisfied. When an edge is fired the discrete variables are updated and a new location is reached. To ensure progress, a clock invariant can be defined on locations. Such an invariant ensures that the automaton does not stay at a location indefinitely, i.e. an edge must be fired before the invariant is violated. A typical Uppaal model consists of a network of parallel timed automata. These automata can communicate via synchronization channels and/or shared variables. The state of such a network is defined by the locations of all automata, the values of the discrete variables, and the constraints on clocks.

Figure 8 shows a small network of parallel timed automata, taken from the Uppaal tutorial [9], that illustrates most of the modelling language constructs. The lamp automaton is depicted on the left and consists out of the locations `off`, `dim`, `bright` and a local clock `c`. The user automaton, depicted on the right, consists out of one initial location and one transition that models the lamp button being pressed. Both automata use the global synchronization channel `press` for communication. The initial location of the lamp is `off`. If the user synchronises with `press!` the lamp moves to the location `dim` and the clock is reset to zero. To set the lamp to `bright` the user has to synchronise again within five time units. Only then is the guard `c<5` satisfied and can the location `bright` be reached. In all other cases the guard `c>=5` is satisfied and the lamp moves to the location `off`. When the lamp is in the location `bright` the user can turn off the lamp by pressing the button once.

In the simulator users can experiment with models by manually choosing the edges that should fire. This enables the user to debug the model and get a general idea of how consistent the be-

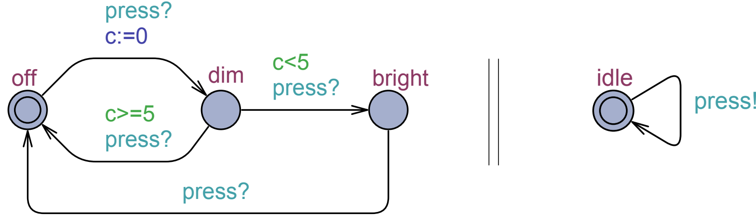


Figure 8: Lamp example

havior is with respect to the requirements. The formal consistency checking with respect to the requirements is done with the model checking engine. When a counter example of a property is found the path is visualised in the simulator.

The UPPAAL model checking engine uses a subset of the Timed Computation Tree Logic (TCTL) language [5, 9] to query the state space. The language consists of path and state formulae. Path formulae quantify over paths and states in the state space. A denotes that a property should hold for all paths, whereas E denotes that there should be at least one path where the property holds. To quantify over the states within a path the symbols [] (globally) and <> (eventually) are used. [] denotes that all states within a path should satisfy the property, whereas <> denotes that at least one state in the path has to eventually satisfy the property. State formulae are properties that can be checked against a state. The following properties are supported by UPPAAL:

- $A[]\phi$ - ϕ must be satisfied for all states (Safety).
- $E[]\phi$ - there exists a path on which all states satisfy ϕ (Safety).
- $E <> \phi$ - there exists a path in which ϕ is eventually satisfied (Reachability).
- $A <> \phi$ - for all paths there eventually must be a state that satisfies ϕ (Liveness).
- $\phi - - > \psi$ - whenever ϕ is satisfied all sequent paths will eventually satisfy ψ (Liveness).

where ϕ and ψ are state formulae.

There are several extensions of UPPAAL, each focusing on different problem domains. For example, UPPAAL TIGA is used for solving games based on timed game automata [8], and UPPAAL CORA is used for cost optimal reachability analysis [10].

4 From TPPO to UPPAAL

In this section a translation from TPPO to UPPAAL is introduced. This translation generates an UPPAAL model that captures the same causal ordering of tasks as defined by the input TPPO. With these generated models the advantages of the UPPAAL model checking engine can be exploited. That is, the model checking engine can efficiently unfold the UPPAAL model of a TPPO by generating the state space. This state space can then be investigated by defining queries as described in section 3. Although UPPAAL is designed for verification of timed systems the translation described in this section does not contain any timing. This is because a TPPO describes the ordering of tasks and does not contain any timed behavior. In section 6 the translation is extended with resources that add time. In this section we completely ignore the notion of resources.

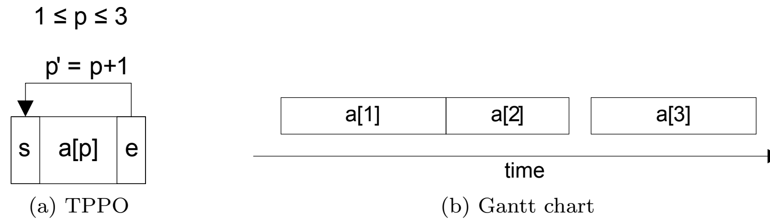
The global idea of the translation is that each task in a TPPO is translated to a task automaton and that events are translated to edges of such a task automaton. An guard is added to each event edge that ensures that an event can only occur when all precedence relations have been met.

The result is a network of task automata where each task has a start and end event edge with a guard. We use the following notation when defining the translation. If X and Y are sets we write $X \rightarrow_* Y$ for the set of partial functions from X to Y. If $f \in X \rightarrow_* Y$ and $x \in X$ we write $f(x) \downarrow$ if $f(x)$ is defined and $f(x) \uparrow$ if $f(x)$ is undefined.

4.1 Efficient Translation

As mentioned earlier, UPPAAL is designed to exhaustively explore the dynamic behavior of a modelled system. Despite the fact that the model checker does this very efficiently the problem of state space explosion is still present. The generated models are only useful if they provide some degree of scalability. To enable a translation that generates scalable models we assume two restrictions on the input TPPO.

The *first restriction* is that each task class has a precedence relations from its *end* to its *start* event. This self loop eliminates auto-concurrency. In other words, a new instance of a task can only occur if the immediate predecessor instance of the same class has finished. Figure 9a shows this dependency for task a, where the *next* instance is defined by the condition $p' = p + 1$. The Gantt chart in figure 9b shows a possible execution sequence for the TPPO in figure 9a. Instead of creating one automaton per task instance this restriction enables efficient reuse of a single task automaton for several instances. After a task instance has finished we can reuse the automaton for the *next* instance by assigning a new valuation to its instantiation variables. To satisfy this



(a) TPPO (b) Gantt chart

Figure 9: Example where new instances depend on the predecessors

restriction we add the requirement that there is a precedence relation between the end and start event of the same task class. That is, for all $t \in T$, $(end(t), start(t)) \in P$. The conditions on these precedence relations are called *next* functions and return the new valuation of the instantiation variables given the current valuation. Figure 9a depicts a precedence relation with a *next* function. For each task $t \in T$, $next_t \in V(t) \rightarrow_* V(t)$. Initially I assigns to each task $t \in T$ an *initial valuation* $I(t) \in V(t)$. We define $R(t)$ to be the least set of valuations that contains $I(t)$ and satisfies $v \in R(t) \wedge next_t(v) \downarrow \;\Rightarrow\; next_t(v) \in R(t)$. Valuations in $R(t)$ are called *reachable*. We define $<_t$ to be the least transitive relation on $R(t)$ satisfying $v <_t next_t(v)$. We require that $<_t$ is irreflexive, that is, there exists no $v \in R(t)$ such that $v <_t v$.

The *second restriction* is that all precedence conditions are partial monotonic functions. These functions are called *update* functions because they map a valuation of the source event to an valuation of the next target event that may occur, e.g. $p' = p + 1$ returns the valuation of the target event p' by increment the valuation of the source event p. These functions are called monotonic because they maintain the order. Given that precedence conditions are monotonic functions one can efficiently derive if all dependencies for an event have been met. That is, instead of maintaining a history of all past event instance one can derive the history from the last event instance that occurred. This decreases the amount of memory needed for verification because the model does not need to maintain the complete history. Thus, for each precedence $p = (A, B) \in P$, an condition also called an update function $U(p) \in V(task(A)) \rightarrow_* V(task(B))$. We write $A \xrightarrow{f} B$ if $(A, B) \in P$ and $U(A, B) = f$. We require, for each task $t \in T$, $U(start(t), end(t)) = \lambda v \in V(t).v$ and $U(end(t), start(t)) = next_t$. Moreover, we require the following monotonicity properties, for

each precedence $A \xrightarrow{f} B$ and $v, w \in V(task(A))$:

$$v <_{\mathsf{task}(A)} w \ \wedge \ f(v)\uparrow \ \Rightarrow \ f(w)\uparrow$$
$$v <_{\mathsf{task}(A)} w \ \wedge \ f(w)\downarrow \ \Rightarrow \ f(v)\downarrow \ \wedge \ f(v) <_{\mathsf{task}(B)} f(w)$$

4.2 Translation

The Uppaal model consists out of a network of finite automata. For each task class $t \in T$ an automaton as depicted in figure 10 is created. The network has a global array M that contains

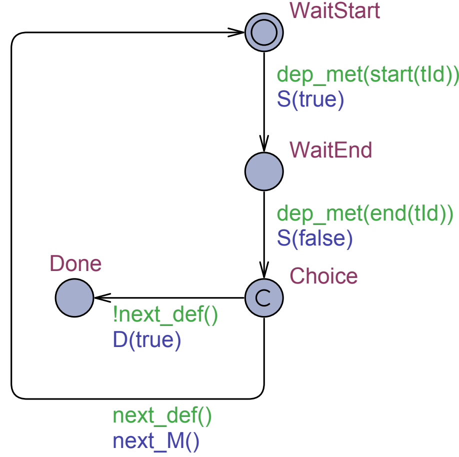


Figure 10: Task automaton

the instantiation variables for each task class. A task instance is defined by the valuation of its instantiation variables. For each $t \in T$, the variables in $M[t]$ are initialised to $I[t]$. Edges are used to model the start and end event of a task. The start event is modelled by the edge between the locations *WaitStart* and *WaitEnd*. The edge between *WaitEnd* and *Choice* models the end event. It is important to note that the automaton ensures that the start event of a task always precedes the end event. Event edges can only be taken when all causal dependencies have been satisfied. The guard *dep_met* checks if all dependencies for the start or end event are satisfied. Here the functions *start* and *end* are used to return the corresponding event given a task id.

Some additional information is stored to keep track of the current location of an automaton. This is needed because in Uppaal one cannot directly determine the current location of an automaton. Therefore, the model maintains two arrays S and D of Boolean variables. For each task $t \in T$, a Boolean variable $S[t]$ records whether the last event that occurred was $end(t)$ or $start(t)$. When an event edge is taken the value of $S[t]$ is set to *true* for the start event and $false$ for the end event. Initially, $S[t] = false$. For each task $t \in T$, a Boolean variable $D[t]$ records whether a task is done. Initially all variables in D are set to false. When all instances for task t have occurred $D[t]$ is set to *true*. In figure 10 the function S and D are used to set the value of the Boolean variables.

When the end event occurs the automaton moves to the location labelled *Choice*. This is a committed location denoted by the C. In such a location no time may pass and the next edge taken must be an outgoing edge of a committed location. The outgoing edges of the committed location determine the next valuation of the instantiation variables. The guard *!next_def* is satisfied when all tasks instances have occurred and the automaton can move to the location *Done*. If still some instances have to occur for task t the guard *next_def* is satisfied. On this edge the instantiation variables are updated by *next_M* that sets $M[t]$ to $next(t)$. The automaton then moves to the location *WaitStart* where it waits for the start event of the new task instance.

For event $e \in E$, the function $eval(e)$ denotes the valuation of the next event instance of e that will occur. The next valuation of e depends on the valuation of the associated task t and on whether the start event has occurred. If no such valuation exists, that is when all event instances of e have occurred already, $eval(e)$ is undefined and we write $eval(e)\uparrow$.

$$eval(e) = \begin{cases} M(t) & \text{if } e = \text{start}(t) \wedge \neg S[t] \\ next(M(t)) & \text{if } e = \text{start}(t) \wedge S[t] \\ M(t) & \text{if } e = \text{end}(t) \end{cases}$$

For an event $b \in E$, the guard $dep_met(b)$ is given by

$$dep_met(b) \quad = \quad \forall (a \xrightarrow{f} b) \in P^- : eval(a)\downarrow \;\wedge\; f(eval(a))\downarrow \quad \Rightarrow \quad eval(b) < f(eval(a))$$

Here P^- is the set of precedences minus all precedences between the start and end event of a task, i.e. $P^- = P \setminus \{(start(t), end(t)) \mid t \in T\}$. These internal task precedences are removed because the task automaton already takes care of these dependencies. The intuition behind the definition of $dep_met(b)$ is as follows. In order to check whether the next event instance of b may occur, we have to check for each incoming precedence $(a \xrightarrow{f} b) \in P^-$ whether the dependency induced by that specific precedence has been met. In order to decide whether the dependency induced by precedence $a \xrightarrow{f} b$ is met, we first test if $eval(A)$ is defined. If this is not the case then all event instances of a have occurred and all dependencies induced by $a \xrightarrow{f} b$ have been met. If still some event instance of a has to occur, we check whether f is defined for the next instance of a. If f is not defined then, by the first monotonicity condition, again all dependencies induced by $a \xrightarrow{f} b$ have been met. If f is defined for $eval(a)$, then we require that the next instance of b precedes $f(eval(a))$. Clearly, if $f(eval(a)) \leq eval(b)$ then the occurrence of the next event instance of a should precede the occurrence of the next event instance of b, and so the dependencies have not been met. Conversely, if $eval(b) < f(eval(a))$ then for any immediate predecessor of $eval(b)$, that is, for any $v \in V(task(a))$ with $f(v) = eval(b)$, the second monotonicity condition implies $v < eval(a)$. Hence, all precedences for $eval(b)$ have been met.

5 Implementation

In the previous section a translation from TPPO to Uppaal is given. This section describe the implementation of this translation. First a textual representation for TPPOs is described that will serve as input for the translator. Then the implementation of the translator is discussed. A overview of the translation phases is given together with a description of the code generation algorithm. The running example will be used to illustrate the implementation.

5.1 TPPO Representation/Syntax

For the implementation of the translator we need a textual representation for TPPOs that will server as input. Within the Octopus project the name Design Space Exploration Intermediate Representation (DSEIR) is used to denote the textual format that describes, among other things, the TPPOs. It is important to note that the DSEIR format presented here is an early version and that the format has changed in newer versions. There are no fundamental differences between the old and new format and therefore it should be relatively easy to adapt the translator to this new format. DSEIR is based on the Y-chart approach (see section 1) and thus describes the tree aspects *application*, *platform*, and *mapping*. The DSEIR file format is based on XML. XML is a textual format used to structure data according to a formal syntax. The syntax of a DSEIR file is defined in a Document Type Definition (DTD) [32] that can be found in Appendix A. Figure 11 shows the DSEIR XML file for the running example. To facilitate a better explanation the example has been divided into *Scenario*, *Application*, *Task*, and *Precedence*. The full DSEIR for

the running example can be found in appendix B.

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <!DOCTYPE scenario SYSTEM "dseir.dtd">
3 <scenario>
4   <platform>
5       ..
6   </platform>
7   <mapping>
8       ..
9   </mapping>
10  <application>
11      ..
12  </application>
13 </scenario>
```

(a) Scenario

```
1 <application>
2   <job>
3     <id>example</id>
4     <range>
5       <id>p</id>
6       <lBound>1</lBound>
7       <uBound>2</uBound>
8     </range>
9     <tasks>..</tasks>
10    <precedences>..</precedences>
11  <job>
12 </application>
```

(b) Application

```
1 <tasks>
2   <task>
3     <id>a</id>
4     <instantiationVar>
5       <id>p</id>
6       <iniValue>1</iniValue>
7     </instantiationVar>
8   </task>
9   <task>
10    <id>b</id>
11    <instantiationVar>
12      <id>p</id>
13      <iniValue>1</iniValue>
14    </instantiationVar>
15  </task>
16 </tasks>
```

(c) Tasks

```
1 <precedences>
2   <precedence>
3     <source>a_s</source>
4     <target>b_s</target>
5     <condition>p'=p</condition>
6   </precedence>
7   <precedence>
8     <source>a_e</source>
9     <target>b_e</target>
10    <condition>p'=p</condition>
11  </precedence>
12  <precedence>
13    <source>b_e</source>
14     <target>a_s</target>
15     <condition>p'=p+1</condition>
16  </precedence>
17 </precedences>
```

(d) Precedences

Figure 11: Running example DSEIR

Scenario
Figure 11a shows the listing of the XML root and its children. Line 1 defines the XML version and the character encoding used. On line 2 the DOCTYPE declaration refers to the external DSEIR DTD. The root of every DSEIR file is scenario. This node contains the three Y-chart aspects, i.e. platform (lines 4-6) describing the resources, mapping (lines 7-9) describing how applications are executed on the platform, and application (lines 10-12) describing the tasks that make up an application. Note that sub-trees mapping and platform are not shown in figure 11. These will be discussed in section 6. In this section we will focus on the application sub-tree.

Application
The application sub-tree contains one or more instances of applications, called jobs. A job is an application that has concrete values for the job parameters. For example, in the domain of digital document printers, the application print in combination with the specific amount of pages to be print is called a job. Each job is described by one TPPO. If an application contains more than one job these can be executed concurrently, i.e. there are no precedence relations between jobs. In figure 11b only one job is defined for the running example (lines 2-11). This job has a unique id at line 3 and defines a range for the instantiation variable p at lines 4-8. The tasks and precedences

sub-trees are shown in figure 11c and 11d.

Task
The tasks sub-tree contains the tasks that are involved in realising a job. Each task has a unique identifier and one or more instantiation variables. Figure 11c shows the listing for the two tasks used in the running example. Each task is identified by an id at line 3 and 10. The instantiationVar nodes contain the id of the used instantiation variable and the node iniValue contains the initial value.

Precedences
The precedence relations between tasks are defined in the precedences sub-tree. For each precedence the source and target events are defined. An event id consists out the task id extended with _s or _e for the start or end event of that task. The condition defines when a precedence relation exists. In figure 11d p' denotes the instantiation variables of the target and p denotes the instantiation variables of the source.

5.2 Translator

The translation process consists out of four phases that gradually transform a DSEIR file to an Uppaal model file. Figure 12 depicts the input and output of each phase. The four phases Parse, Convert, Code generation, and Serialise are depicted by arrows that transform some input to an output. Here the documents DSEIR XML and Uppaal XML denote the external input and output files of the translator. Internally the translator uses the trees DSEIR, IR, and NTA as input and output of the phases.

The translator is implemented in the Java programming language. Java was chosen because of its widespread use, portability, and extensive open source community. The parse and serialise phase are implemented with the XStream open source library. This library provides an user friendly interface for mapping XML tags to Java objects and vice versa. Given such a mapping, XStream automates the parsing from XML and the serialisation to XML. The tool is available at http://xstream.codehaus.org/.

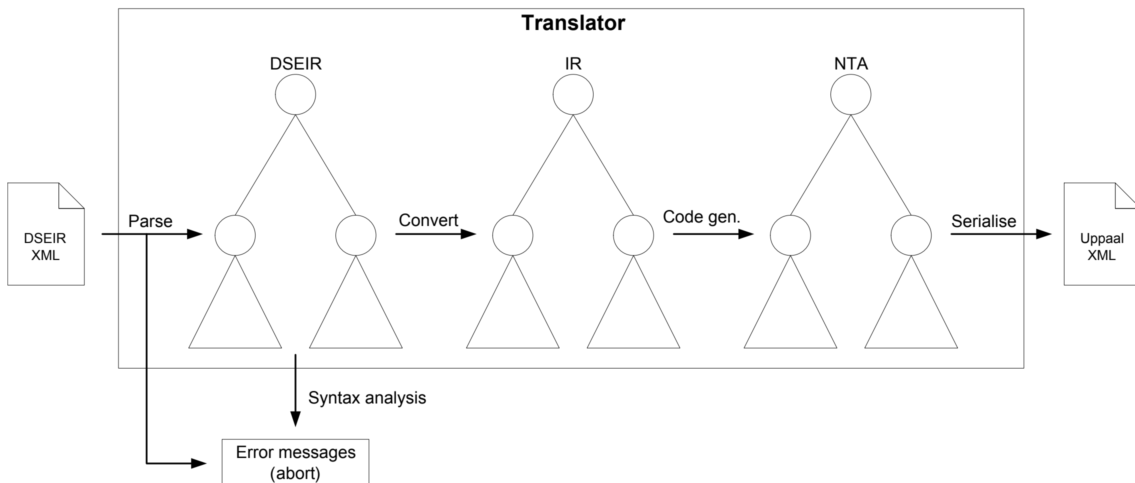


Figure 12: Translation phases

Parse & Syntax analysis
The parse phase has two goals. The first goal is to determine if the structure of the DSEIR XML file is correct with respect to the grammar given in appendix A. The second goal is to transform

the input file to a tree structure that provides easy access to the data. The strings that makeup a XML document can be divided into markup and content. Markup strings, also known as tags, describe the structure of the document, whereas the content is the actual data we wish to store. Note that the parsing process only checks the markup syntax and not the syntax of the content strings. Some additional analysis is done after the parse phase to check the syntax of the content. For example, an error is thrown when the source or target of a precedence relation is undefined.

The implementation of the parse phase uses the XStream library. The DSEIR tree structure is modelled as a set of Java classes. The relationships between these classes describe the allowed structure of the tree. There is a one-to-one mapping between the tags in the DSEIR XML file and the classes of the DSEIR tree in Java. The only exception being that the XML may contain implicit collections that have to be modelled explicitly in Java, e.g. with an array or list. Given this mapping XStream can automatically instantiate a DSEIR tree from a DSEIR XML file. If the structure of the XML file cannot be parsed an exception is thrown giving the location of the error in the XML file.

Convert
The converter phase transforms a DSEIR tree into an IR (Intermediate Representation) tree. The goal is to transform and rearrange the input tree so that it is suited for the code generation phase. The addition of an intermediate representation simplifies the implementation of the code generation phase. For example, in the DSEIR data structure there is a clear separation between application, mapping, and platform. During code generation this separation requires extra navigation through the data structure. Therefore the convert phase creates a new data structure that combines application and platform information by applying the mapping. Another advantage of the convert phase and IR is that it decouples the code generation phase from DSEIR. This makes the translator less sensitive to changing in the DSEIR data structure. If a converter can be implemented for the changed DSEIR data structure no changes to the code generation phase are necessary. When such a conversion is not possible the IR and the code generation have to be adapted.

Code generation
In this phase a Network of Timed Automata (NTA) is generated from the IR. The output of this phase is a NTA tree that represents the generated UPPAAL model. Such a tree consists of automata templates and code. The task template, as depicted in figure 10, is predefined and can simply be added to the NTA tree. The bulk of the work in this phase deals with the generation of code that implements the logic behind the precedence relations. Sections 5.3 gives an overview of the code generation algorithm.

Serialise
The last phase serialises the NTA tree to the XML format used by UPPAAL. The formal syntax of such a UPPAAL XML file is defined in a DTD that is available at http://www.it.uu.se/research/group/darts/uppaal/flat-1_1.dtd. The serialisation of the NTA tree is implementation with the XStream library. There is a one-to-one mapping between the classes in the NTA tree and tags in the XML file. Given this mapping XStream can automatically serialise a NTA tree to a UPPAAL XML file.

5.3 Code Generation

Based on the translation given in 4 we now describe the algorithm that generates the UPPAAL code. The input of this translation is a DSEIR XML file that descibes a TPPO. The output is code that implements the TPPO precedence dependencies. Each task and event is assigned an non-negative integers that serves as a unique identifier. To enable a mapping from tasks to events and back the encoding $start(t) = 2t$, $end(t) = 2t + 1$, $task(t) = t/2$ is used. Listing 1 shows the implementation of these functions.

```
1 int start(int id){return id*2;}
2 int end(int id){return id*2+1;}
3 int task(int id){return id/2;}
```

Listing 1: Task and event encoding

The algorithm is broken down into nine small steps, each step building on the previous step(s). To clarify the algorithm examples of generated code are given for the running example in figure 11.

1. *Identifiers*: For each task class and event unique identifiers are generated. These identifiers are used throughout the model and will reappear in almost every piece of generated code. The total number of task classes are stored in a constant variable called nTasks. In line 3 of listing 2 the type tId_T is defined as an integer with a lower bound of 0 and an upper bound of nTask−1. A value between these bounds identify a task class. For each task class a constant variable is generated that is assigned an unique value of type tId_T (see lines 4-5). The same scheme is applied for generating the event identifiers at lines 8-13 . Note that the number of events in line 8 is nTasks*2 as each task has a start and end event denoted by _s and _e.

   ```
   1 // Task identifiers
   2 const int nTasks = 2;
   3 typedef int[0,nTasks-1] tId_T;
   4 const tId_T a_id = 0;
   5 const tId_T b_id = 1;
   6
   7 // Event identifiers
   8 const int nEvents = nTasks*2;
   9 typedef int[0,nEvents-1] eId_T;
   10 const eId_T a_s = 0;
   11 const eId_T a_e = 1;
   12 const eId_T b_s = 2;
   13 const eId_T b_e = 3;
   ```

   Listing 2: Identifiers for the running example

2. *Range*: Each task class has a set of instantiation variables. A valuation of these variables defines a task instance. To limit the number of task instances a range is given that defines the upper and lower bounds for each instantiation variable. Lines 1-3 show the code that is generated from the range of the running example. The constant variables lBound_p and uBound_p are used in other pieces of generated code to check whether a valuation is within bounds. For implementation reasons some instantiation variables have to be marked as undefined. For this purpose a unique value is assigned to undef_p. An instantiation variable p is said to be undefined if its value equal the value of undef_p.

   ```
   1 const int lBound_p = 1;
   2 const int uBound_p = 2;
   3 const int undef_p = lBound_p-1;
   ```

   Listing 3: Range for the running example

3. *Instantiation variables*: As mentioned each task has a set of instantiation variables. Current versions of UPPAAL do not support a set data structure. Therefore sets are modelled as structs that group together several variables. Lines 1-4 show the definition of such a struct named varSet_T. For each instantiation variable a variable with the same name is added to the struct definition. In the case of the running example there is only one instantiation variable p at line 3. The function undef at lines 6-10 can be used to check whether all variable values in a set are within the range. In line 11 an array M of varSet_T instances is generated.

Each index in this array contains the set of instantiation variables for a specific task. All task instantiation variables are assigned the initial value.

```
typedef struct
{
  int p;
} varSet_T;

bool undef(const varSet_T &v)
{
  return (v.p < lBound_p && v.p != undef_p) || v.p > uBound_p;
}

varSet_T M[tId_T] ={{1},{1}};
```

Listing 4: Instantiation variables for the running example

4. *Next function*: As described in section 4, each task has a partial function called *next* that returns the new valuation of its instantiations variables given the current valuation. All the *next* functions are modelled in a single UPPAAL function `next`, that accepts a task `id` and returns the new valuation of the instantiation variables. A switch is generated using cascading `if else` statements. For each task a case is added to the switch that calculates the next valuation (see lines 3-14). For the running example the next valuation is the increment of `p`. As mentioned all *next* functions are partial functions, i.e. not every input maps to and output. Therefore an extra function called `next_def` is generated that returns whether the update function is defined for some input. For each task a case is added that checks whether the next valuation falls within the bounds (see lines 20-24).

```
varSet_T next(tId_T id)
{
  varSet_T newSet;
  if(id == a_id)
  {
    newSet = M[id];
    newSet.p++;
  }
  else
  if(id == b_id)
  {
    newSet = M[id];
    newSet.p++;
  }
  return newSet;
}

bool next_def(tId_T id)
{
  if(id == a_id)
    return M[id].p < uBound_p;
  else
  if(id == b_id)
    return M[id].p < uBound_p;
}
```

Listing 5: Next function for the running example

5. *Precedence function*: The precedence relations are modelled in a single function `P`. It accepts two events `A`, `B` and returns whether a precedence exists between these two events. A switch is generated using cascading `if else` statements. For each precedence relation a case is added

that returns true. If no precedence relation exists between A and B the function will return false.

```
1  bool P(eId_T A, eId_T B)
2  {
3    if(A == a_s && B == b_s)
4      return true;
5    else
6    if(A == a_e && B == b_e)
7      return true;
8    else
9    if(A == b_e && B == a_s)
10     return true;
11   else
12     return false;
13 }
```

Listing 6: Precedence function for the running example

6. *Update function*: Each precedence condition is assumed to be a partial monotonic function (see section 4). All update functions are modelled in a single function U, that accepts a source event A, a target event B, and the valuation of the source event v. A switch is generate using cascading if else statements. For each precedence relation a case is added to the switch that applies the update function and returns the new valuation (see lines 3-19). As mentioned all updates are partial functions. Therefore an extra function called U_def is generated that returns whether the update function is defined for some input. First the update function is applied on the given arguments (line 24). Then the undef function at line 25 determines if the resulting output is defined, i.e. if all instantiation variables are within the range.

```
1  varSet_T U(eId_T A, eId_T B, varSet_T v)
2  {
3    if(A == a_s && B == b_s)
4    {
5      v.p=v.p;
6      return v;
7    }
8    else
9    if(A == a_e && B == b_e)
10   {
11     v.p=v.p;
12     return v;
13   }
14   else
15   if(A == b_e && B == a_s)
16   {
17     v.p=v.p+1;
18     return v;
19   }
20 }
21
22 bool U_def(eId_T A,eId_T B,varSet_T v)
23 {
24   varSet_T uv =U(A,B,v);
25   return !undef(uv);
26 }
```

Listing 7: Update function for the running example

7. *Location identifiers*: In UPPAAL one cannot directly determine the current location of a task automaton from within a function. Some additional data has to be maintained to enable this. Each task automaton maintains one boolean variable in the arrays S and D. Here S[t] records whether the last event instance of task t was a start or end event and D[t] records whether all instances of task t have occurred. If we relate this to the locations of the task automaton in figure 10, then S[t]==false denotes $WaitStart$, S[t]==true denotes $WaitEnd$, and D[t]==true denotes *Done*. Initially all Booleans are set to false as this corresponds to the initial locations of the task automata.

```
1 bool S[tId_T] = {false, false};
2 bool D[tId_T] = {false, false};
```

Listing 8: Location identifiers for the running example

8. *Eval function*: For each event the partial function eval gives the next event instance that will occur. The next instance or valuation of an event depends on the current location of the associated task automaton, maintained in the arrays S and D. If event e is a start event and the current location is $WaitEnd$ then the next event valuation is the next valuation of the associate task (see line 7). Otherwise the location is $WaitStart$ and the next event valuation is the current task valuation (see line 9). If event e is an end event the next event valuation is always the current task valuation (see line 12). To model the fact that eval is a partial function eval_def is generated. eval is undefined if the next task instance is undefined (line 21) or if the task automaton is in the location *Done* (line 23 and 26). In all other cases eval is defined.

```
1  varSet_T eval(eId_T e)
2  {
3    tId_T t = task(e);
4    if(e == start(t))
5    {
6      if(S[t])
7        return next(t);
8      else
9        return M[t];
10   }
11   else
12     return M[t];
13 }
14
15 bool eval_def(eId_T e)
16 {
17   tId_T t = task(e);
18   if(e == start(t))
19   {
20     if(S[t])
21       return next_def(t);
22     else
23       return !D[t];
24   }
25   else
26     return !D[t];
27   }
```

Listing 9: Eval function for the running example

9. *dependencies met function*: The guard dep_met checks if all dependencies for an event are satisfied. This function builds upon the previously generated code.

```
1 bool dep_met(eId_T B)
2 {
3   return forall (A : eId_T) P(A,B) imply
4     (eval_def(A) && U_def(A,B,eval(A)) imply eval(B).p < U(A,B,eval(A)).p);
5 }
```

Listing 10: Dependencies met function for the running example

6 Resources

In the previous sections we explained how TPPOs are used to describe applications and how these can be translated to Uppaal. In this section we extend the translation with resources that determine the execution time. Applications are executed on a platform consisting of various resources, e.g. RAM and hard disks for storage, buses for data transport, FPGA blocks and general purpose CPUs for processing. Each resource has a limited capacity, e.g. max memory or bus bandwidth. The total capacity of a resource is divided into a number of chunks that can be claimed or released. Resources are used to execute tasks that make up an application and are typically claimed at the start and released at the end of a task. Depending on the available capacity a claim may succeed or fail. A task can only start executing if all resource claims have succeeded. When a task finishes it typically releases the claimed resources. However, in some cases instead of releasing a resource it is handed over to another task. When the task, to which the resource was handed over, finishes the resource is released or handed over to yet another task. An common example is when one task hands over memory that contains data for another task. In this research we modelled resource handovers implicitly, i.e. when a task does not release a claimed resource it is assumed that eventually another task will release it. Another solution is to explicitly define to which task a resource is handed over. In this solution we know exactly which task will use and release the resource. An disadvantage of explicit resource handover is that its implementation requires addition logic, whereas implicit resource handover does not require any additional logic. The duration of a task depends on the size of the work and the pace at which the claimed resources can process the work. If the pace is constant the duration of a task is defined by $size/pace$. However, during execution the pace of a resource may change resulting in slower or faster task execution. Contention between tasks for resources occurs because resources are limited and multiple tasks can be executed concurrently. Scheduling policies are used to resolve resource contention and attempt to optimize, for example, throughput, turnaround time, and response time. In practise these scheduling policies are based on heuristic algorithms and thus are not guaranteed to find the optimal solution. For the generated Uppaal models no scheduling policies need to be implemented as the model checking engine can explore the whole state space and find the optimal solution. However, the computation time and memory usage can increase sharply due to state space explosion. This problem can be alleviated by implementing some scheduling policies that decreases the nondeterminism in the model, with the risk of losing the optimal solution. Another advantage of scheduling policies is that the model behavior comes closer to the behavior of a real system.

The running example used in the previous sections describes a situation in which task a writes some data into a buffer which is then processed by task b. Task a and b are pipelined, which means that b can start before a has finished. We extend the running example with a simple platform consisting of a general purpose dual core CPU and 32 MB of RAM. Both tasks are mapped on the CPU and are assigned at least one core. If two cores are available both will be used doubling the pace of a task. All the 16 MB buffer memory is claimed by a and is only released when b has finished. In this simple example we abstract for the buses that transport data from one resource to another. Figure 13 shows the TPPO and the resource mapping for the running example. Here ellipses denote resources, edges from resource to task denote claims, and edges from task to resource denote releases. The amount that is claimed or released is shown in a label next to the edge.

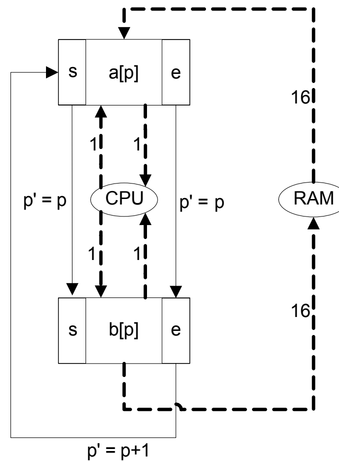


Figure 13: Running example with resources and mapping

6.1 Resource Representation/Syntax

In sections 5.1 we described the textual representation of TPPOs. This section describes the textual representation of resources and the mapping that maps tasks on resources. Figure 14 shows the DSEIR XML file for the running example. Note that only the platform and mapping sub-trees are shown. The application sub-tree was already discussed in section 5.1. To facilitate a better explanation the example has been divided into *Scenario*, *Platform*, *Map task a*, and *Map task b*. The full DSEIR for the running example can be found in appendix B.

Scenario
Figure 14a shows the listing of the XML root and its children. The root contains the three Y-chart aspects, i.e. platform (lines 4-6) describing the resources, mapping (lines 7-9) describing how applications are executed on the platform, and application (lines 10-12) describing the tasks that make up an application.

Platform
A platform consists of one or more resources. Each resource has a capacity that gives the number of chunks that can be claimed or released by a task. The pace of a resource is defined by the pace function that returns the speed at which a resource can process data. Figure 14b shows the platform node that contains the resources for the running example. The dual core CPU is modelled as one resource that has a capacity of two (line 3), i.e. one chunk for each core. The paceFunction node, at lines 4-7, describes the dynamic behavior of the CPU. Here the Uppaal expression syntax is used, which is almost identical to C. At a later stage it may be desirable to use an intermediate expression language that can be translated to the languages of the used tools. For now we define expressions in the language of the target tool, in this case Uppaal. The resource_cap array maintains the state of each resource. If all cores have been claimed the pace is 10. If one core is idle it will be used to double the pace of the task that has claimed the other core. Memory is modelled as a single resource in lines 9-13. The total capacity of the memory resource is 32 MB (line 11) and the pace is a high constant (line 12).

Mapping
The mapping node describes how the tasks are mapped onto the platform. For each task it contains a map node that describes the size of the work and the resources that are claimed and released. Figure 14c shows the mapping for task *a*. The size node (line 4) gives the size of the data that has to be processed. To resolve resource contention a *lazy* or *greed* claiming strategy is defined for each task (line 5). If the node claimStrategy is not defined the lazy strategy will be used as default.

A greedy strategy can be used to reduce the state space or to accurately model a system with greedy scheduling. Note that with a greedy strategy the optimal solution may be lost. Before task a can start it has to claim one CPU core (lines 6-9) and a 16 MB buffer (lines 10-13). When a is done the claimed CPU core is released (lines 14-17). The buffer is not released by task a because it serves as input for task b that may still be busy. The mapping for task b is shown in 14c. The size of the data that b has to process is larger than task a (line 4). Another difference is that b does not claim any memory as it uses the buffer claimed by a. When b is done it releases the buffer and the CPU core (lines 10-17).

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <!DOCTYPE scenario SYSTEM "dseir.dtd">
3  <scenario>
4    <platform>
5        ..
6    </platform>
7    <mapping>
8        ..
9    </mapping>
10   <application>
11       ..
12   </application>
13 </scenario>
```

(a) Scenario

```
1  <resource>
2    <id>cpu</id>
3    <capacity>2</capacity>
4    <paceFunction>
5      if (resource_cap[cpu_id]==0) return 10;
6      else return 20;
7    </paceFunction>
8  </resource>
9  <resource>
10   <id>mem</id>
11   <capacity>32</capacity>
12   <paceFunction>return 999;</paceFunction>
13 </resource>
```

(b) Platform

```
1  <map>
2    <jobId>example</jobId>
3    <taskId>a</taskId>
4    <size>100</size>
5    <claimStrategy>lazy</claimStrategy>
6    <claim>
7      <resourceId>cpu</resourceId>
8      <capacity>1</capacity>
9    </claim>
10   <claim>
11     <resourceId>mem</resourceId>
12     <capacity>16</capacity>
13   </claim>
14   <release>
15     <resourceId>cpu</resourceId>
16     <capacity>1</capacity>
17   </release>
18 </map>
```

(c) Map task a

```
1  <map>
2    <jobId>example</jobId>
3    <taskId>b</taskId>
4    <size>160</size>
5    <claimStrategy>lazy</claimStrategy>
6    <claim>
7      <resourceId>cpu</resourceId>
8      <capacity>1</capacity>
9    </claim>
10   <release>
11     <resourceId>mem</resourceId>
12     <capacity>16</capacity>
13   </release>
14   <release>
15     <resourceId>cpu</resourceId>
16     <capacity>1</capacity>
17   </release>
18 </map>
```

(d) Map task b

Figure 14: DSEIR platform and mapping for the running example

6.2 Extending the Network of Timed Automata

To enable resource usage the network is extended with an automaton, as depicted in figure 16, for each resource. Such a resource automaton maintains the current capacity of a resource. In order to claim and release resources the task automata are extended with a one-way message passing mechanism that enables communication from one task to multiple resources. To keep track of a tasks duration, each task automaton is extended with a clock variable and a mechanism that handles changes in resource pace. Another option is to add clocks to each resource automata as was done in [20]. In this solution a resource keeps track of its duration and signals a task automaton when it is done. This simplifies the mechanism for detecting changes in resource pace as this can

be handled within a single resource automaton. The disadvantage of this approach is that each resource requires multiple clocks when several tasks are using the resource concurrently, i.e. one clock to maintain the progress of each task. To avoid complex resource automata we choose to extend the task automata with clocks. Figure 15 depicts an extended task automaton.

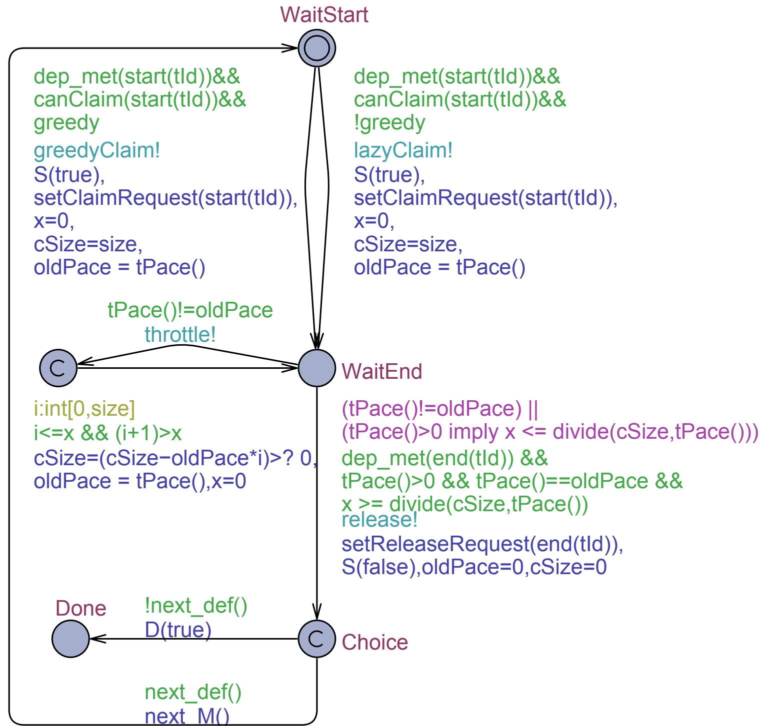


Figure 15: Task automaton with resource usage

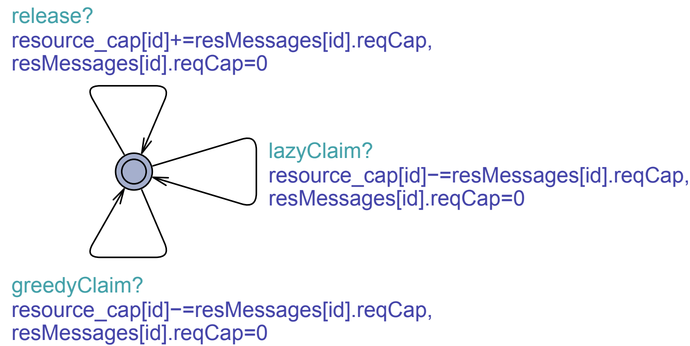


Figure 16: Resource automaton

Claiming/Releasing Resources

To enable the claiming and releasing of resources task and resource automata pass data via channels and shared variables. The general idea is that a task sets a message in a shared variable and uses a channel to notify the resource of this new message. The channels used to notify resources are so called broadcast channels. With broadcast channels one sender automaton can synchronise with zero or more receiver automata. This enables one task automaton to set multiple variables

and notify all resource automata. After receiving a notification the resources will check if they received a message.

To avoid state space explosion the model can be fine-tuned by choosing the appropriate scheduling strategy for claiming resources. For each task the claiming strategy can be set to either *lazy* or *greedy*. With the lazy strategy the claiming of resources can be postponed even if the requested resources are available, thus postponing the execution of a task as it can only start when all resources have been claimed. This introduces a lot of nondeterminism increasing the state space. On the other hand this will ensure that the optimal solution is found. When a greedy strategy is used a task is forced to claim the required resources as soon as they become available. This decreases the nondeterminism and thus reduces the state space, with the risk of losing the optimal solution. Note that there is still some nondeterminism left when two or more tasks want to claim the same resource at the same time. In future versions priorities could be added to resolve this nondeterminism and enable further fine-tuning of the model.

In figure 15 the two edges between locations $WaitStart$ and $WaitEnd$ denote the start event of a task. These edges can only be taken when the guards *dep_met* and *canClaim* are satisfied. The guard *dep_met* was already discussed in sections 4 and 5. The guard *canClaim* is satisfied when all the required resources for the task are available. The two edges that denote the start event differ in the used claiming strategy. The left edge is taken when a task uses a greedy strategy, i.e. *greedy* is satisfied, and the right edge is used for the lazy strategy, i.e. !*greedy* is satisfied. When a greedy strategy has been chosen the urgent broadcast channels *greedyClaim* is used for communication between task and resources. Declaring a channel urgent ensures that no delay can occur if a synchronization is enabled. For lazy resource claims the non-urgent broadcast channel *lazyClaim* is used that does allow delays. Before synchronizing with the resources the update function *setClaimRequest* sets a variable to the requested capacity for each resource. These variables are stored in the global array $resMessage$ that uses the unique resource identifiers as indexes. When the variables are set the task automaton synchronises with the resource automata. During synchronization a resource automaton takes one of the edges labelled with the *greedyClaim* or *lazyClaim* synchronization. Given its unique identifier a resource can look up its current capacity, stored in the global array *resource_cap*, and update it by subtracting the requested capacity stored in the array $resMessage$. The last update resets the variable used for message passing. The same schema is used to release resources. The edge between locations $WaitEnd$ and $Choice$ denotes the end event of a task and the *release* channel on this edge synchronises with the resources. The invariant on the location $WaitEnd$ ensures that the end event edge cannot be delayed and thus no delay can occur when releasing a resource, i.e. all resource releases are considered to be greedy.

Task Duration

The duration of a task is defined by the time that elapses between the occurrence of the start and end event. This task duration depends on the size of the work and the pace at which the claimed resources can process the work. If the pace is constant the duration of a task is defined by *size*/*pace*. However, during execution the pace of one or more resources may change resulting in faster or slower task execution. In the task automaton shown in figure 15 the elapsed time since the last occurrence of a start event is maintained by clock x. The function *tPace* gives the current pace at which the task can process its work. This is defined by the resource that forms the bottleneck, i.e. the resource that has the lowest pace. The time at which a task will end is given by $divide(cSize, tPace())$ [1]. Here the variable *cSize* is the current size of the work that still has to be processed by the task.

Time elapses when the task is in the location $WaitEnd$. To force progress the invariant $tPace()! = oldPace \mid\mid (tPace() > 0\ imply\ x <= divide(cSize, tPace())$ defines when this location may be oc-

[1] In UPPAAL we can only impose integer bounds on clock variables. Therefore the function $divide(a, b)$ gives the smallest integer greater or equal to $\frac{a}{b}$.

cupied. The global idea is that the elapsed time, given by clock x, should not exceed the task duration given by $divide(cSize, tPace())$. To avoid violation of the invariant the end event edge should be taken when both the invariant and the guard are satisfied, i.e. $x == divide(cSize, tPace())$.

To model dynamic resources we need to know exactly when the pace of a resource changed. When such a change occurs the guard $tPace()! = oldPace$ is satisfied. The urgent property of the broadcast channel *throttle* ensures that the automaton moves to the committed location without delay. Because there are no receivers *throttle* will synchronise with zero automata. In order to calculate the work that has been done we multiply the value of clock x with $oldPace$. However, Uppaal cannot refer to the value of clocks in integer assignments. Therefore we use a trick, described in [20], to derive the largest integer i that satisfies $i \leq x$. This trick uses the select statement $i : int[0 : size]$ that non-deterministically binds i to a value in the range 0 to $size$. The guard $i <= x \ \&\& \ (i+1) > x$ ensures that i is the largest integer that satisfies $i \leq x$. The value of $cSize$ can then be updated to $(cSize - OldPace * i) >?0$. Here $>?$ is the maximum operator that ensures that the new value of $cSize$ is at least 0. The last step is to update $oldPace$ and reset x. It is important to note that rounding down the clock value leads to a small over approximation of the amount of work that still has to be done.

6.3 Code Generation

We now describe the algorithm that generates the code needed for resource usage. The algorithm is broken down into seven small steps, each step building on the previous step(s). To clarify the algorithm examples of generated code are given for the running example in figure 14.

1. *Identifiers*: For each resource an unique identifier is generated. The total number of resources are stored in the constant variable nResources. In line 2 the type rId_T is defined as an integer with a lower bound of 0 and an upper bound of nResources−1. A value between these bounds identifies a resource. For each resource a constant variable is generated that is assigned an unique value of type rId_T (see lines 4-5).

```
1 const int nResources = 2;
2 typedef int[0,nResources-1] rId_T;
3
4 const rId_T cpu_id = 0;
5 const rId_T mem_id = 1;
```

Listing 11: Resource identifiers for the running example

2. *Message passing code*: Task automata communicate with resource automata via channels and shared variables. When a greedy strategy has been chosen the urgent broadcast channels greedyClaim is used for communication between task and resources (line 1). For lazy resource claims the non-urgent broadcast channel lazyClaim is used (line 2). Resources can be released with the non-urgent broadcast channel release (line 3). The shared variables used to pass messages from task to resource are of type resMessage_T (lines 5-8). Such a struct contains a field with the requested capacity of a resource. More fields can be added to extend the resource message. Instances of resMessage_T are stored in the global array resMessage that uses the resource identifiers as indexes (line 9). The meta keyword is used to denote that resMessage is not part of the state when doing model checking, i.e. two states are considered to be equal when they only differ in meta variables. The throttle channel, at line 11, is used for its urgent property. It ensures that a task automaton throttles its pace without delay.

```
1 urgent broadcast chan greedyClaim;
2 broadcast chan lazyClaim;
3 broadcast chan release;
4
5 typedef struct
6 {
```

```
7   int reqCap;
8 } resMessage_T;
9 meta resMessage_T resMessages[rId_T];
10
11 urgent broadcast chan throttle;
```

Listing 12: Message passing code for the running example

3. *Resource capacity*: The current capacity of each resource is maintained in the array resource_cap. Initially each index is set to the maximum capacity of the corresponding resource. This state information can be used to throttle the pace of a resource.

```
1 int resource_cap[rId_T] = {2,32};
```

Listing 13: Resource capacity for the running example

4. *Resource pace function*: For each resource the rPace function returns the current pace at which it can operate. A switch is generated using cascading if else statements. For each resource a case is added to the switch that applies the paceFunction of the DSEIR input file. In the example below we can clearly see the code contained in the paceFunction nodes of figure 14b.

```
1 int rPace(rId_T id)
2 {
3   if(id==cpu_id)
4   {
5     if (resource_cap[cpu_id]==0) return 10;
6     else return 20;
7   }
8   else
9   if(id==mem_id)
10  {
11    return 999;
12  }
13 }
```

Listing 14: Resource pace function for the running example

5. *canClaim function*: Before a start event can occur the guard canClaim has to be satisfied. A switch is generated from the DSEIR claim nodes that returns whether the require resources can be claimed.

```
1 bool canClaim(int id)
2 {
3   if(id==a_s)
4     return resource_cap[cpu_id]>=1 && resource_cap[mem_id]>=16;
5   else
6   if(id==b_s)
7     return resource_cap[cpu_id]>=1;
8   else
9     return true;
10 }
```

Listing 15: canClaim function for the running example

6. *Set message functions*: The function setClaimRequest sets the message variables to the requested capacity (lines 1-13). For each *start* event a case is added that sets the requested capacities. To release resources setReleaseRequest sets the message variables to the capacity that should be released (lines 15-27). For each *end* event a case is added that sets the capacities to the amount that should be released.

```
1 void setClaimRequest(int id)
2 {
3   if(id==a_s)
4   {
5     resMessages[cpu_id].reqCap = 1;
6     resMessages[mem_id].reqCap = 16;
7   }
8   else
9   if(id==b_s)
10  {
11    resMessages[cpu_id].reqCap = 1;
12  }
13 }
14
15 void setReleaseRequest(int id)
16 {
17   if(id==a_e)
18   {
19     resMessages[cpu_id].reqCap = 1;
20   }
21   else
22   if(id==b_e)
23   {
24     resMessages[mem_id].reqCap = 16;
25     resMessages[cpu_id].reqCap = 1;
26   }
27 }
```

Listing 16: Set message functions for the running example

7. *Task pace function*: The function tPace returns the current pace at which a task can process its work. A tasks pace is defined by the resource that forms the bottleneck, i.e. the resource that has the lowest pace. For each task a case is generated that returns the minimum pace from the list of claimed resources. Here <? is the minimum operator.

```
1 int tPace(int id)
2 {
3   if(id==a_id)
4   {
5     return rPace(cpu_id) <? rPace(mem_id);
6   }
7   else
8   if(id==b_id)
9   {
10    return rPace(cpu_id);
11  }
12 }
```

Listing 17: Task pace function for the running example

7 Comparison

In the previous sections we presented an implementation for generating UPPAAL models from an abstract intermediate representation (DSEIR). In this section we compare these generated models to manually constructed models. For this comparison we use an industrial case study from Océ technologies and the UPPAAL models that were manually constructed for it. The case study and models are described in [20]. The goal is to verify that the behavior of the generated models

corresponds to the manually constructed models. Other important aspects in the comparison are memory usage and computation timed during model checking.

7.1 The Case Study

We first introduce the case study, which has been taken from [20, 4]. This case study describes an experimental setup of a digital printer/copier. It is important to note that the performance numbers used in the case study do not represent the performance of any real system. The main challenge in the case study is to compute efficient schedules for jobs that minimise execution time.

A typical multifunctional printer/copier performs a variety of image processing functions on digital documents in addition to scanning, copying and printing. Apart from local use for scanning and copying, users can also remotely use the system for image processing and printing. A generic architecture of the system is shown in figure 17. The system has two ports for input: Scanner and

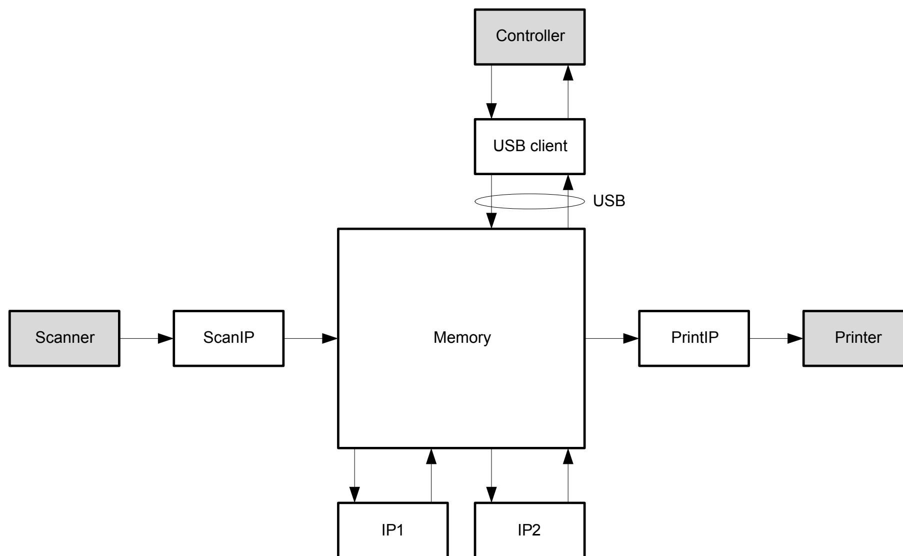


Figure 17: Generic architecture

Controller. Users locally come to the system to submit jobs at the Scanner and remote jobs enter the system via the Controller. These jobs use the image processing (IP) components (ScanIP, IP1, IP2, PrintIP), and system resources such as memory and a USB bus for executing the jobs. Finally, there are two places where the jobs leave the system: Printer and Controller. The IP components can be used in different combinations depending on how a document is requested to be processed by the user. This gives rise to several applications of the system, that is, each job may use the system in a different way. The list of components used by a job defines the datapath for that job. Some examples of applications are:

- **direct copy**: Scanner, ScanIP, IP1, IP2, USBClient,PrintIP
- **scan to store**: Scanner, ScanIP, IP1, USBClient
- **scan to email**: Scanner, ScanIP, IP1, IP2, USBClient
- **process from store**: USBClient, IP1, IP2, USBClient
- **simple print**: USBClient, PintIP
- **print with processing**: USBClient, IP2, PrintIP

It is not mandatory that components in a datapath process a job sequentially, i.e. the design of the system allows for a certain degree of parallelism. The degree of parallelism is defined by the TPPOs of the various applications. Figure 18 shows the TPPO for *direct copy* together with the

resources and mapping. This example illustrates how the components of the generic architecture are used.

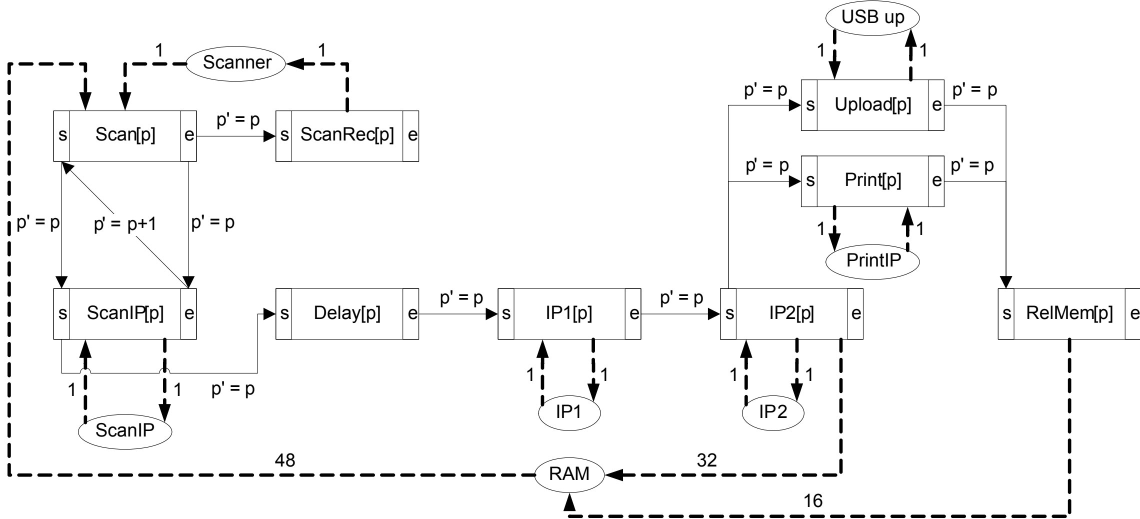


Figure 18: Direct copy TPPO with resources and mapping

In the case study many image processing components are designed to perform a single task, e.g. ScanIP, IP1, and IP2. Therefore, the task and the component claimed by it often have the same name. In figure 18 the tasks Scan and ScanIP process pages in parallel. This is because ScanIP processes the output of Scan on a line-by-line basis (streaming) and has the same throughput as Scan. The Scan task for the next page can only start when ScanIP has ended. Another constraint for the start of Scan is imposed by the fact that the scanner has to recover after scanning a page. This is handled by the task ScanRec that releases the scanner when it is ready for the next page. Only after the scanner is released can it be claimed by the Scan task for the next page. The dependency between tasks ScanIP and IP1 is different. IP1 processes the output of ScanIP in streaming mode and has a higher throughput than ScanIP. Therefore, IP1 may start processing the page in parallel with ScanIP, with a certain delay due to the higher throughput of IP1. This delays is modelled with a task that has a fixed duration. Because of the nature of the image processing function that IP2 performs, IP2 can start processing a page only after IP1 has finished. The output of IP2 servers as input for the tasks Print and Upload that can start processing the output of IP2 immediately. Pages are uploaded to the controller for later use, e.g. pages are downloaded when multiple copies need to be printed.

In addition to the image processing components, two other system resources are memory and USB bandwidth. Processing of a single page is only allowed if the entire memory required for completion is available and allocated. Each task uses a certain amount of memory that is released once the computation has finished and no other task needs the information stored in the memory. In figure 18, Scan claims 48 MB of RAM per page that is gradually released by IP2 and RelMem. The task RelMem has a duration of zero time units and is used to synchronise the release of memory. Only when Print and Upload have finished can the last 16 MB of RAM be released. Another important resource is the USB. This bus has limited bandwidth and serves as a bridge between the USBClient and the memory. The bus may be used both for uploading and for downloading data. At most one task may upload data at any point in time, and similarly at most one task may download data. Uploading and downloading may take place concurrently. If only one task is using the bus, transmission takes place at a rate of *high* MB/s. If two tasks use the bus then transmission takes place at a slightly lower rate of *low* MB/s. This is referred to as *dynamic* USB behavior. Approximately, low is 75% of high. The reason why it is not 50% is that the USB

protocol also sends acknowledgment messages, and the acknowledgment for upward data can be combined with downward data, and vice versa. The *static* USB behaviour is one in which the transmission rate is always high MB/s. To model the dynamic behavior the USB is spilt into an *USB up* and *USB down* resource. When the two resources are used concurrently the transmission rate for both will drop to low.

7.2 Manually Constructed Models

For the comparison we use Uppaal models that were manually constructed for the case study. The basic idea is that each application is represented by a single automaton that describes the order in which resources are used. In the manually constructed models the term use-case is used instead of application. Figure 19 shows the automaton for the *simple print* use-case. A single sequence from the location *INIT* to *DONE* represents one page undergoing several image processing tasks. Use-case automata synchronise with resource automata depicted in figure 20. Initially a resource is in the location *IDLE*. When a use-case automaton claims a resource it enters the *RUNNING* location. How long a resource remains in this location depends on the variable *execution_time* that is compared with clock *x*. After *execution_time* units have elapsed the resource moves to the location *RECOVERING* where it stays for *recover_time* units. When *recover_time* units have elapsed the resource automaton signals the use-case automaton that it can continue. Note that here each resource automaton contains a clock that maintains the elapsed time. In the generated models resources do not contain clocks. Instead, each task automaton contains a clock.

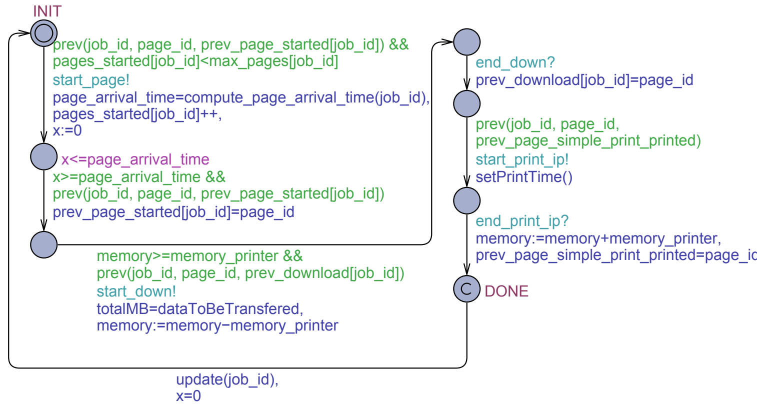


Figure 19: Simple print template

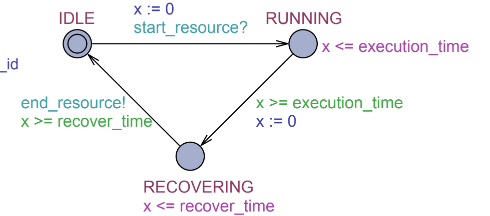


Figure 20: Resource template

The manually constructed models described above are slightly modified versions of the models described in [20]. To improve performance an edge is added from *DONE* to *INIT* that enables reuse of use-case automata. Another improvement is the addition of a nonovertaking rule that ensures that pages of the same job cannot overtake each other, i.e. the second page of a job cannot be processed before the first page of the same job.

7.3 Experiments

The case study introduces a generic printer platform and applications that can be executed on this platform. For the experiments we use two platforms that only differ in the used USB. In the first platform a static USB is used that always has the same transmission rate. The second platform uses a dynamic USB that can throttle its transmission rate depending on whether uploading and downloading is taking place concurrently or not. Several different applications are mapped onto these two platforms. By combining different applications instances, called jobs, and platforms we can define test scenarios that will serve as benchmarks for the comparison in section 7.4. All

experiments were performed using Uppaal version 4.1.2. To avoid interference from other non-Uppaal processes all experiments were performed on a Sun Fire X4440 server with 16 cores (AMD Opteron 8356, 2.3GHz) and 128 GB of DDR2 RAM. Because Uppaal version 4.1.2 is limited to a single processor and a maximum of 4GB of memory the performance should be almost identical on a desktop pc with a comparable CPU and 4GB of RAM.

The Formats '08 benchmark is taken from [20] and consists of a *process from store* job of three pages, a *print with processing* job of two pages, a *scan to email* job of one page, and a *scan to store* job of one page. For this scenario we use the platform with static and dynamic USB and 96 MB of RAM. This benchmark contains a lot of resource contention because several jobs, that require the same resources, are executed concurrently. Table 1 shows an overview of the job arrival times and the required resource per page for each job.

Application	**Arrival Time**	**Required Resources Per Page**	**Pages**
Process from store	0	24 MB RAM, USB (down), IP1, IP2, USB (up)	3
Print with processing	0	12 MB RAM, USB (down), IP2, PrintIP	2
Scan to email	1	48 MB RAM, Scanner, ScanIP, IP1, IP2, USB (up)	1
Scan to store	1	36 MB RAM, Scanner, ScanIP, IP1, USB (up)	1

Table 1: Formats '08 benchmark

The scenario described above is a good benchmark for concurrent jobs. However, in practice the number of pages per job are higher and the number of jobs that are executed concurrently on a printer/copier is smaller. Therefore we define single job benchmarks, with varying number of pages, for the applications *simple print*, *process from store*, *scan to store*, and *direct copy*. We also define two benchmarks in which two jobs, with varying number of pages, are executed concurrently. These two benchmarks use the compound application *direct copy & simple print* and *direct copy & process from store*. To ensure continuous interleaving the applications with a higher throughput, i.e. *simple print* and *process from store*, are assigned more pages than *direct copy* that has a lower throughput. For the single and concurrent jobs we only consider a platform with dynamic USB as this is the more interesting behavior. Table 2 shows an overview of the single and concurrent benchmarks. Here the number of pages is a range from 1 to 100. Several experiments are performed with values from this range. To push the Uppaal model checker to its limits two additional benchmarks, with a higher number of pages, will be performed for *direct copy & simple print* and *direct copy & process from store*.

Application(s)	**Arrival Time**	**Required Resources Per Page**	**Pages**
Simple print	0	12 MB RAM, USB (down), PrintIP	1-100
Process from store	0	24 MB RAM, USB (down), IP1, IP2, USB (up)	1-100
Scan to store	0	36 MB RAM, Scanner, ScanIP, IP1, USB (up)	1-100
Direct copy	0	48 MB RAM, Scanner, ScanIP, IP1, IP2, USB (up), PrintIP	1-100
Direct copy & Simple print	0 0	48 MB RAM, Scanner, ScanIP, IP1, IP2, USB (up), PrintIP 12 MB RAM, USB (down), PrintIP	1-100 1-100
Direct copy & Process from store	0 0	48 MB RAM, Scanner, ScanIP, IP1, IP2, USB (up), PrintIP 24 MB RAM, USB (down), IP1, IP2, USB (up)	1-100 1-100

Table 2: Single and concurrent benchmarks

For each experiment we are interested in the precise behavior that results in an optimal scheduling solution. Given a model, Uppaal can find an optimal solution and generate a trace file for it. From such a trace file a Gantt chart is generated that visualises the start and end time for each task. By comparing the Gantt charts we can verify whether the behavior of the generated models corresponds to the manually constructed models. Appendix C shows the Gantt charts with static and dynamic USB for the manual and generated models. Besides model behavior, we are also interested in the performance of the Uppaal model checker during verification. For each experiment the following performance metrics are used: peak memory usage, running time, size of the generated trace file, and the number of states explored during verification. The latency of the fastest schedule is used as an indicator of the model behavior and should be equal for both models. Appendix D contains the results of the experiments.

7.4 Results

By comparing the Gantt charts for the Formats '08 benchmarks we can verify whether the behavior of the generated models corresponds to the manual models. Appendix C contains the Gantt charts for the comparison. First we compare the static USB charts of the manual and generated models. The charts for both models are identical, i.e. the start and end time for each task is identical. Figure 21 show this Gantt chart. In the second comparison we look at the dynamic USB charts of the manual and generated models. When we compare the manual model chart in figure 22 with the generated model chart in figure 23 a small difference in the USB *down* task of the *print from store* (pfs) job can be seen. In the manual model the *down* task for the first page is delayed one time unit, whereas in the generated model no such delay occurs. Delaying this task does not effect the latency of the schedule. In fact there are several optimal solutions from which Uppaal picks one. Because the manual and generated models use a different structure Uppaal picks another optimal solution. The start and end times for all other tasks are identical. For the remaining experiments we will not show the Gantt charts as this will require too much space. Instead we will use the latency of the found solution as an indicator of model behavior.

We now continue with the comparison of the Uppaal model checker metrics. Table 3 shows the experiment for the Formats '08 benchmark. We can observe that the number of states explored for the generated models is significantly reduced compared to the manual models. However, the running time for the generated models does increase compared to the manual models. This is mainly due to some inefficiencies in the generate code. For example, the generated code depends heavily on cascading *if else* statements as Uppaal does not support the *switch* construct. We recommend extending the Uppaal language with an optimized switch, e.g. using branch table optimization. This is a useful language extension and would reduce the running time of the generated models.

USB Behavior	Model	Peak Mem Usage(KB)	Running Time(s)	Latency (s)	Size Trace	States Explored
Static	Manual	37876	25.51	22	334K	400661
	Generated	20696	43.73	22	288K	64222
Dynamic	Manual	35272	23.91	25	347K	369673
	Generated	30556	103.66	25	312K	192014

Table 3: Formats '08 Experiments

More experiments for single and concurrent applications, with varying number of pages, can be found in tables 4 and 5 of appendix D. In total 41 experiments were performed that confirm the observations made in table 3, i.e. the generated models perform better in terms of states explored and the manual models perform better in terms of running time. If we look at the concurrent

applications in table 5 we observe that the generated models have the advantage of a lower peak memory when a high number of pages are used.

Both models use variables to keep track of the current page numbers. These variables are monotonically increased and thus define a measure of progress. This property can be used to reduce memory usage during model checking. The basic idea is that, depending on the progress of the model, some states can never be reached again and thus can be deleted from the history of visited states. A more detailed description of this method can be found in [13]. Tables 6 and 7 in appendix D show the experiments with progress measures. We observe a decrease in peak memory and running time for both manual and generated models. For the generated models with progress measures we can verify the extreme benchmarks *direct copy* & *simple print* for 1400 & 1930 pages and *direct copy* & *process from store* for 450 & 900 pages. Without progress measures these extreme benchmarks cause an out of memory exception during model checking.

8 Conclusion and Future Work

In this master thesis, we presented an approach for generating Uppaal models from a high-level representation specifying an embedded system under development. These generated models help designers to explore possible design options, i.e. the models can be used to get more insight into the chosen design options which can then be improved. We have addressed the research questions formulated in section 1. Answering these research questions resulted in: (i) a formal definition of task-level parameterized partial orders; (ii) a translation from task-level parameterized partial orders to Uppaal; (iii) an extended translation with resources; (iv) a comparison between manually constructed and generated models.

The use of parameterized partial orders, developed within the Octopus project, is a novel approach for specifying compact task graphs with repetitive behavior. We have given a definition of Task-level Parameterized Partial Orders based on three steps. First we define the classical model of Event-level Partial Orders (EPOs). In the second step we extend this definition to Parameterized Partial Orders (EPPOs) that compactly represent repetitive events. In the third and last step we defined Task-level Parameterized Partial Orders (TPPOs) that pairs start and end events to create tasks. An important advantage is that TPPOs have a well defined definition that is based on the well known classical model of EPOs. By lifting the notation to the task level we create a more intuitive representation, i.e. engineers typically specify system activities in terms of tasks.

All model checking tools face the problem of state space explosion, that must be addressed to solve most nontrivial problems. To elevate this problem we have defined an efficient translation based on two assumption on the input TPPO, i.e. tasks are not auto-concurrent and all precedence conditions are monotonic functions. For the case study used in this research these assumptions are perfectly valid. However, in other cases these assumptions may restrict the generated models too much. For example, it may be necessary to allow auto-concurrency. In future work, this can be accomplished by defining a translation that creates an automaton for each task instance. Before translating a TPPO one could automatically check whether to use the efficient or less efficient translation.

By extending the translation with a platform consisting of various resources we add timing to the generated models. Typically resources that execute a task are claimed at the start and released at the end of the task. The implicit resource handovers used in this research assume that when a task does not release a claimed resource another task eventually will. An improvement would be to add an explicit handover mechanism that keeps track of the tasks to which resource are handed over. This is especially important if a resource that is handed over determines the execution speed of a task. The duration of a task depends on the pace of the claimed resources. Some resources have dynamic behavior that may result in faster or slower task execution, de-

pending on how many tasks are using the resource concurrently. A complex example of dynamic behavior are buses that continuously have to throttle their speed. We have modelled a somewhat simplified version of bus throttling with the dynamic USB. Here a function is used to derive the speed from the current state. It should be relatively straight forward to model more complex cases of bus throttling by extending the logic of these functions. However, it is likely that more state information has to be maintained to derive the speed for the more complex cases.

We have implemented the translation in the Java programming language and applied it to an Océ case study that focuses on the digital data path of a printer/copier. The challenge in the case study is to compute efficient schedules for jobs that minimise execution time. The results were compared to manually constructed Uppaal models created for the same case study. For the comparison a total of 43 experiments were performed for several different scenarios. When comparing the behavior of the models we sometimes observed small differences in the behavior that do not effect the end times of the schedules. These differences occur because there can be several optimal schedules from which one is picked. Because the manual and generated models have a different structure Uppaal sometimes picks another solution. Other important aspects in the comparison are peak memory usage and computation timed during model checking. Here the number of states explored has a big influence on peak memory. During model checking the states explored for the generated models is roughly half of that of the manual models. This result in a lower peak memory for the more complex test scenarios, e.g. scenarios with a lot of concurrency and a high number of pages. With the reduce in peak memory we were able to verify two extreme scenarios that could not be verified by the manually constructed models because of an out of memory exception. However, in terms of computation time the generated models are outperformed by the manually constructed models. This is mainly due to some inefficiencies in the generated code. For example, the generated code depends heavily on cascading *if else* statements as Uppaal does not support the *switch* statement. The addition of an optimized switch statement in the Uppaal language would reduce the computation time as it could replace the inefficient cascading *if else* statements.

Because Octopus is an ongoing project we expect that some of the future work discussed here will be applied in newer versions of the framework. In general we conclude that the generated Uppaal models are most suited for the early phases of design space exploration. Here nondeterminism may be used to reflect uncertainties in the design, e.g. no scheduling policies have to be specified. The disadvantage of this approach is that it aggravates the state space explosion problem. As the number of uncertainties decrease a more deterministic model can be created and simulation may be more appropriate than model checking.

References

[1] Yasmina Abdeddaïm, Eugene Asarin, and Oded Maler. Scheduling with timed automata. *Theoretical Computer Science (TCS)*, 354(2):272–300, 2006.

[2] Yasmina Abdeddaïm, Abdelkarim Kerbaa, and Oded Maler. Task graph scheduling using timed automata. *Parallel and Distributed Processing Symposium (IPDPS'03), International*, 0:237b, 2003.

[3] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman, editors. *Compilers: principles, techniques, and tools*. Pearson/Addison Wesley, Boston, MA, USA, second edition, 2007.

[4] Israa AlAttili, Fred Houben, Georgeta Igna, Steffen Michels, Feng Zhu, and Frits W. Vaandrager. Adaptive scheduling of data paths using uppaal tiga. In S. Andova et.al., editor, *First Workshop on Quantitative Formal Methods: Theory and Applications (QFM'09), Eindhoven, The Netherlands*, pages 1–12. Electronic Proceedings in Theoretical Computer Science 13, 2009.

[5] Rajeev Alur, Costas Courcoubetis, and David Dill. Model-checking for real-time systems. In *Proc. 5th IEEE Symposium on Logic in Computer Science (LICS90)*, pages 414–425. IEEE Computer Society Press, 1990.

[6] Rajeev Alur and David L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126:183–235, 1994.

[7] Andrew W. Appel and Jens Palsberg. *Modern Compiler Implementation in Java*. Cambridge University Press, New York, NY, USA, 2003.

[8] Gerd Behrmann, Agnès Cougnard, Alexandre David, Emmanuel Fleury, Kim G. Larsen, and Didier Lime. *Uppaal Tiga User-manual*, 2007. www.cs.aau.dk/∼adavid/tiga/manual.pdf.

[9] Gerd Behrmann, Alexandre David, and Kim G. Larsen. A tutorial on uppaal. In Marco Bernardo and Flavio Corradini, editors, *SFM*, volume 3185 of *Lecture Notes in Computer Science*, pages 200–236. Springer, 2004.

[10] Gerd Behrmann, Kim Guldstrand Larsen, and Jacob Illum Rasmussen. Priced timed automata: Algorithms and applications. In *FMCO*, pages 162–182, 2004.

[11] Johan Bengtsson and Wang Yi. Timed automata: Semantics, algorithms and tools. In Jörg Desel, Wolfgang Reisig, and Grzegorz Rozenberg, editors, *Lectures on Concurrency and Petri Nets*, volume 3098 of *Lecture Notes in Computer Science*, pages 87–124. Springer, 2003.

[12] Thomas Bøgholm, Henrik Kragh-Hansen, Petur Olsen, Bent Thomsen, and Kim G. Larsen. Model-based schedulability analysis of safety critical hard real-time java programs. In *JTRES '08: Proceedings of the 6th international workshop on Java technologies for real-time and embedded systems*, pages 106–114, New York, NY, USA, 2008. ACM.

[13] Søren Christensen, Lars Michael Kristensen, and Thomas Mailund. A sweep-line method for state space exploration. In *TACAS 2001: Proceedings of the 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 450–464, London, UK, 2001. Springer-Verlag.

[14] Alexandre David, Jacob Illum, Kim G. Larsen, and Arne Skou. Model-based framework for schedulability analysis using uppaal 4.1. January 2009.

[15] N. Trcka et al. Parameterized timed partial orders and resources: Formal definition and semantics. Technical Report ESR-2010-02, Eindhoven University of Technology, 2010.

[16] Ansgar Fehnker. Scheduling a steel plant with timed automata. In *the Sixth International Conference on Real-Time Computing Systems and Applications (RTCSA'99)*, pages 280–287, Washington, DC, USA, 1999. IEEE Computer Society.

[17] Martijn Hendriks, Barend van den Nieuwelaar, and Frits Vaandrager. Recognizing finite repetitive scheduling patterns in manufacturing systems. In *ASAP, University of Nottingham, United Kingdom*, pages 291–319, 2003.

[18] Martijn Hendriks, Barend van den Nieuwelaar, and Frits Vaandrager. Model checker aided design of a controller for a wafer scanner. *Software Tools for Technology Transfer (STTT)*, 8(6):633–647, 2006.

[19] Martijn Hendriks and Marcel Verhoef. Timed automata based analysis of embedded system architectures. In *Workshop on Parallel and Distributed Real-Time Systems 2006*. IEEE, 2006.

[20] Georgeta Igna, Venkatesh Kannan, Yang Yang, Twan Basten, Marc Geilen, Frits Vaandrager, Marc Voorhoeve, Sebastian de Smet, and Lou Somers. Formal modeling and scheduling of datapaths of digital document printers. In Franck Cassez and Claude Jard, editors, *FORMATS*, volume 5215 of *Lecture Notes in Computer Science*, pages 170–187. Springer, 2008.

[21] Kurt Jensen and Lars M. Kristensen. *Coloured Petri Nets: Modelling and Validation of Concurrent Systems*. Springer, 2009.

[22] Kurt Jensen, Lars Michael Kristensen, and Lisa Wells. Coloured petri nets and cpn tools for modelling and validation of concurrent systems. *International Journal on Software Tools for Technology Transfer (STTT)*, 9(3-4):213–254, 2007.

[23] Venkatesh Kannan, Lou Somers, and Marc Voorhoeve. Datapath architecture simulation. In M. Al-Akaidi, editor, *Proceedings 23rd European Simulation and Modelling Conference (ESM'2009)*, pages 238–242. Ostend: Eurosis-ETI, 2009.

[24] Bart Kienhuis, Ed Deprettere, Kees Vissers, and Pieter van der Wolf. An approach for quantitative analysis of application-specific dataflow architectures. In *ASAP '97: Proceedings of the IEEE International Conference on Application-Specific Systems, Architectures and Processors*, page 338, Washington, DC, USA, 1997. IEEE Computer Society.

[25] Bart Kienhuis, Ed F. Deprettere, Pieter van der Wolf, and Kees A. Vissers. A methodology to design programmable embedded systems - the y-chart approach. In *Embedded Processor Design Challenges: Systems, Architectures, Modeling, and Simulation - SAMOS*, pages 18–37, London, UK, 2002. Springer-Verlag.

[26] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978.

[27] Michael G. Norman and Peter Thanisch. Models of machines and computation for mapping in multicomputers. *ACM Computing Surveys*, 25(3):263–302, 1993.

[28] Simon Perathoner, Ernesto Wandeler, Lothar Thiele, Arne Hamann, Simon Schliecker, Rafik Henia, Razvan Racu, Rolf Ernst, and Michael González Harbour. Influence of different system abstractions on the performance analysis of distributed real-time systems. In *EMSOFT '07: Proceedings of the 7th ACM & IEEE international conference on Embedded software*, pages 193–202, New York, NY, USA, 2007. ACM.

[29] Wilhelm Reinhard and Maurer Dieter. *Compiler Design*. Addison Wesley, New York, NY, USA, 1995.

[30] Concepcio Roig, Ana Ripoll, and Fernando Guirado. A new task graph model for mapping message passing applications. *IEEE Transactions on Parallel Distributed Systems*, 18(12):1740–1753, 2007.

[31] Thomas A. Sudkamp. *Languages and machines: an introduction to the theory of computer science*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, second edition, 1997.

[32] World Wide Web Consortium (W3C). Extensible Markup Language (XML) 1.0. World Wide Web publication, 2008. http://www.w3.org/TR/xml.

[33] Glynn Winskel. An introduction to event structures. In *Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency, School/Workshop*, pages 364–397, London, UK, 1989. Springer-Verlag.

A DSEIR Document Type Definition (DTD)

```
1  <!--nodes-->
2  <!ELEMENT scenario (platform,mapping,application)>
3    <!ELEMENT platform (resource*)>
4      <!ELEMENT resource (id,capacity,paceFunction)>
5    <!ELEMENT mapping (map*)>
6      <!ELEMENT map (jobId,taskId,size,claimStrategy?,claim*,release*)>
7        <!ELEMENT claim (resourceId,capacity)>
8        <!ELEMENT release (resourceId,capacity)>
9    <!ELEMENT application (job+)>
10     <!ELEMENT job (id,parameter*,range+,userFunction*,tasks,precedences)>
11       <!ELEMENT parameter (id,type,value)>
12       <!ELEMENT range (id,lBound,uBound)>
13       <!ELEMENT tasks  (task+)>
14         <!ELEMENT task (id,instantiationVar+,condition?)>
15           <!ELEMENT instantiationVar (id,iniValue)>
16       <!ELEMENT precedences (precedence*)>
17         <!ELEMENT precedence (source,target,condition)>
18
19   <!--leafs-->
20   <!ELEMENT id (#PCDATA)>
21   <!ELEMENT capacity (#PCDATA)>
22   <!ELEMENT paceFunction (#PCDATA)>
23   <!ELEMENT jobId (#PCDATA)>
24   <!ELEMENT taskId (#PCDATA)>
25   <!ELEMENT size (#PCDATA)>
26   <!ELEMENT claimStrategy (#PCDATA)>
27   <!ELEMENT resourceId (#PCDATA)>
28   <!ELEMENT userFunction (#PCDATA)>
29   <!ELEMENT type (#PCDATA)>
30   <!ELEMENT value (#PCDATA)>
31   <!ELEMENT lBound (#PCDATA)>
32   <!ELEMENT uBound (#PCDATA)>
33   <!ELEMENT condition (#PCDATA)>
34   <!ELEMENT iniValue (#PCDATA)>
35   <!ELEMENT source (#PCDATA)>
36   <!ELEMENT target (#PCDATA)>
```

B DSEIR XML file for the running example

```
<<?xml version = "1.0" ?>
<!DOCTYPE scenario SYSTEM "DSEIR.dtd">
<scenario>
  <platform>
    <resource>
      <id>cpu</id>
      <capacity>2</capacity>
      <paceFunction>
        if (resource_cap[cpu_id]==0) return 10;
        else return 20;
      </paceFunction>
    </resource>
    <resource>
      <id>mem</id>
      <capacity>32</capacity>
      <paceFunction>return 999;</paceFunction>
    </resource>
  </platform>
  <mapping>
    <map>
      <jobId>example</jobId>
      <taskId>a</taskId>
      <size>100</size>
      <claim>
        <resourceId>cpu</resourceId>
        <capacity>1</capacity>
      </claim>
      <claim>
        <resourceId>mem</resourceId>
        <capacity>16</capacity>
      </claim>
      <release>
        <resourceId>cpu</resourceId>
        <capacity>1</capacity>
      </release>
    </map>
    <map>
      <jobId>example</jobId>
      <taskId>b</taskId>
      <size>160</size>
      <claim>
        <resourceId>cpu</resourceId>
        <capacity>1</capacity>
      </claim>
      <release>
        <resourceId>mem</resourceId>
        <capacity>16</capacity>
      </release>
      <release>
        <resourceId>cpu</resourceId>
        <capacity>1</capacity>
      </release>
    </map>
  </mapping>
  <application>
    <job>
```

```
57        <id>example</id>
58          <range>
59            <id>p</id>
60            <lBound>1</lBound>
61            <uBound>2</uBound>
62          </range>
63         <tasks>
64          <task>
65            <id>a</id>
66            <instantiationVar>
67              <id>p</id>
68              <iniValue>1</iniValue>
69            </instantiationVar>
70          </task>
71          <task>
72            <id>b</id>
73            <instantiationVar>
74              <id>p</id>
75              <iniValue>1</iniValue>
76            </instantiationVar>
77          </task>
78        </tasks>
79        <precedences>
80          <precedence>
81            <source>a_s</source>
82            <target>b_s</target>
83            <condition>p'=p</condition>
84          </precedence>
85          <precedence>
86            <source>a_e</source>
87            <target>b_e</target>
88            <condition>p'=p</condition>
89          </precedence>
90          <precedence>
91            <source>b_e</source>
92            <target>a_s</target>
93            <condition>p'=p+1</condition>
94          </precedence>
95        </precedences>
96      </job>
97    </application>
98  </scenario>
```

C Gantt Charts

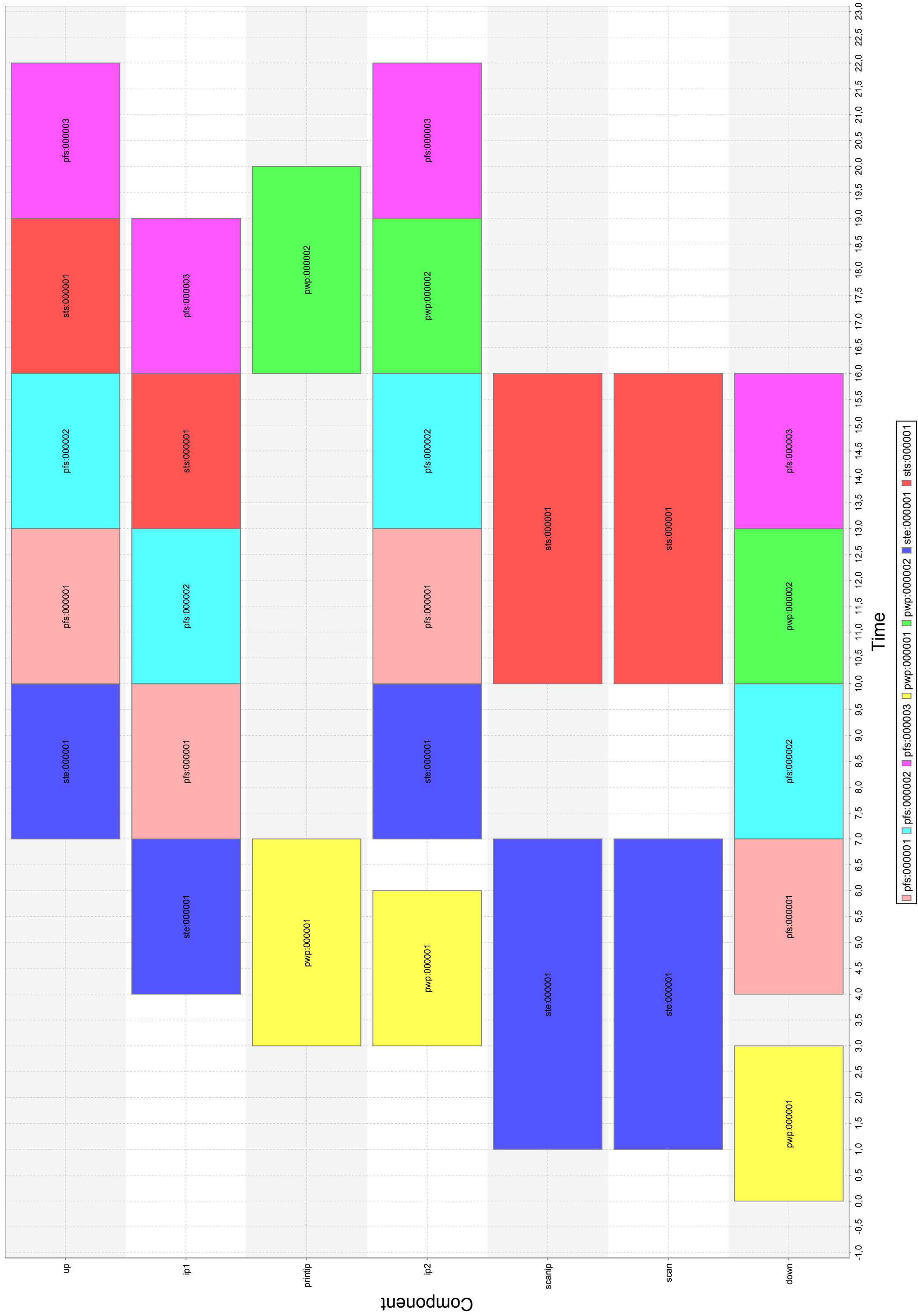


Figure 21: Optimal schedule for manual and generated models with static USB

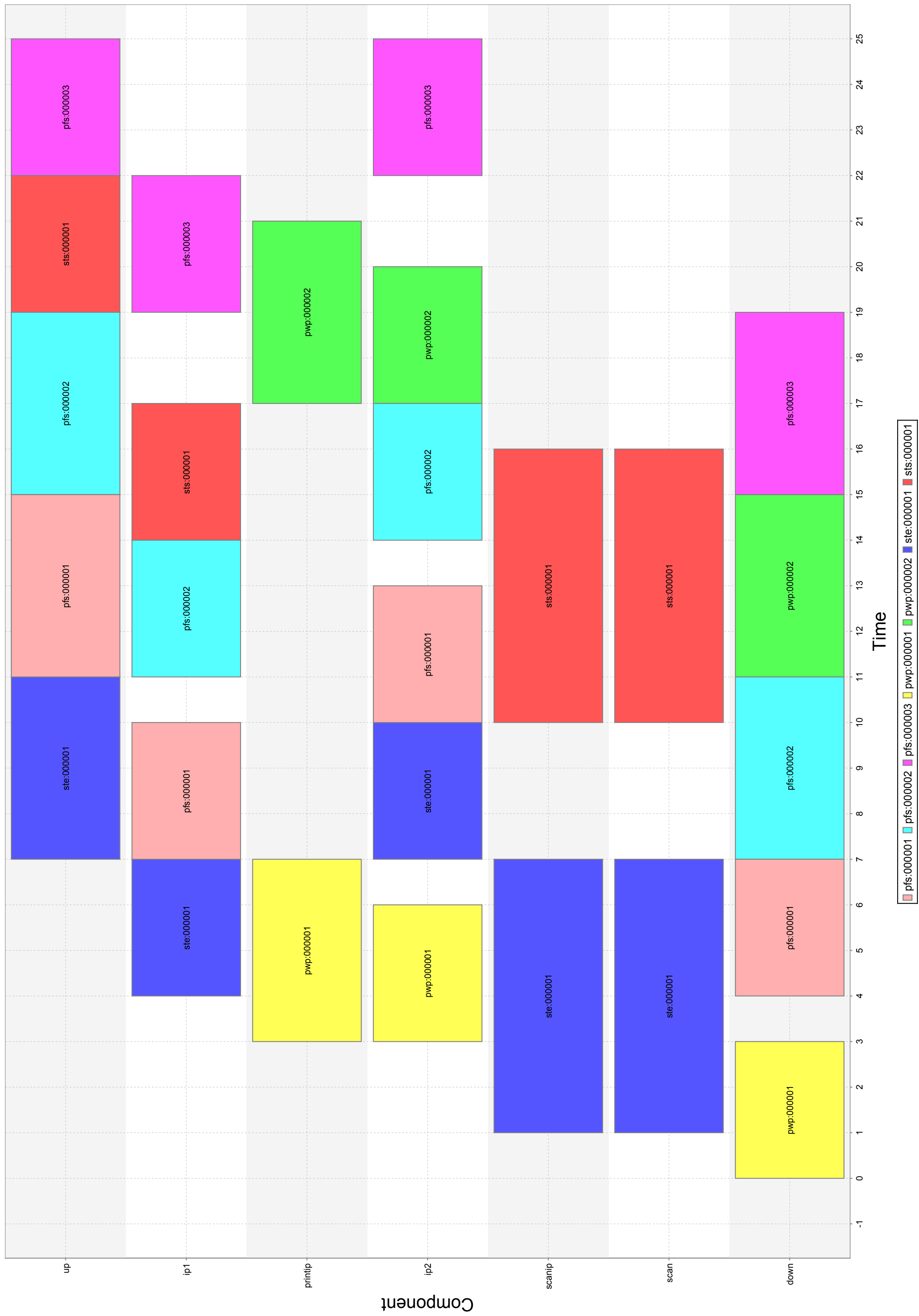


Figure 22: Optimal schedule for manual model with dynamic USB

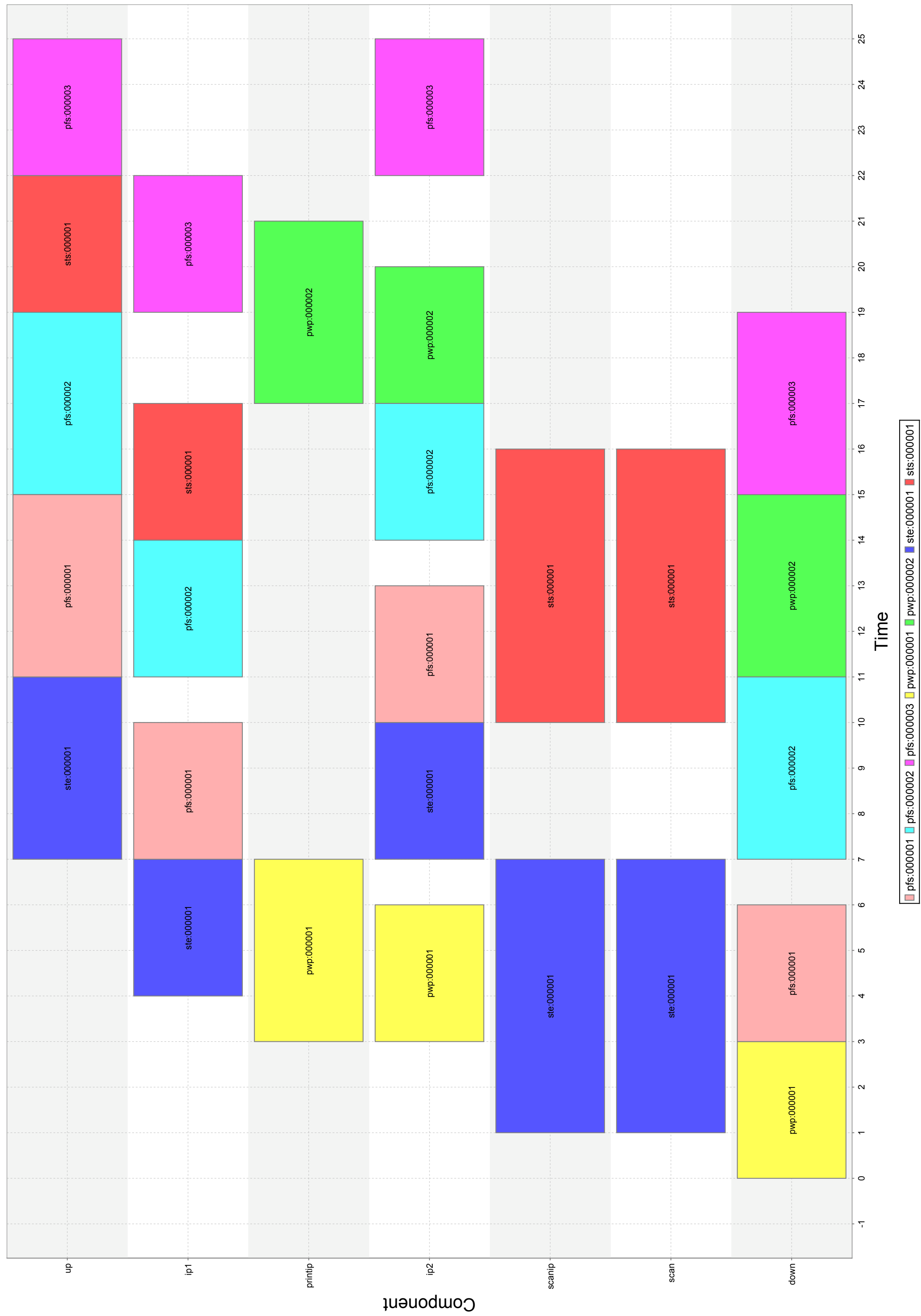


Figure 23: Optimal schedule for generated model with dynamic USB

D Experiments

App.	No Pages	Manual Models					Generated Models				
		Peak Mem Usage(KB)	Running Time(s)	Latency (s)	Size Trace	States Explored	Peak Mem Usage(KB)	Running Time(s)	Latency (s)	Size Trace	States Explored
Simple Print	1	3300	0.30	7	26K	9	136	0.10	7	5.1K	5
	2	3960	0.20	11	49K	20	132	0.10	11	9.7K	10
	5	4028	0.30	23	118K	54	136	0.10	23	24K	24
	10	4140	0.40	43	233K	116	132	0.10	43	46K	50
	20	4344	0.70	83	462K	264	3300	0.20	83	91K	98
	50	4828	1.50	203	1.1M	613	3360	0.30	203	229K	249
	100	5500	2.80	403	2.3M	1163	3456	0.50	403	460K	499
Process from Store	1	3944	0.20	9	44K	18	3684	0.20	9	15K	10
	2	3984	0.20	12	80K	68	3796	0.20	12	27K	31
	5	4088	0.40	24	211K	246	3760	0.20	24	71K	127
	10	4252	0.70	44	428K	656	3868	0.30	44	140K	292
	20	4552	1.20	84	863K	1476	3940	0.50	84	277K	622
	50	5340	2.80	204	2.2M	3936	4144	1.00	204	690K	1612
	100	6520	5.50	404	4.3M	8036	4448	2.00	404	1.4M	3262
Scan to Store	1	3652	0.20	9	35K	33	3136	0.20	9	28K	23
	2	3684	0.20	16	69K	69	3240	0.20	16	54K	45
	5	3768	0.30	37	168K	165	4268	0.30	37	133K	111
	10	3900	0.50	72	334K	325	4340	0.50	72	264K	221
	20	4100	0.90	142	667K	645	4460	0.80	142	527K	441
	50	4636	2.10	352	1.7M	1605	4792	1.80	352	1.3M	1101
	100	5472	4.20	702	3.3M	3205	5292	3.50	702	2.6M	2201
Direct Copy	1	3880	0.20	10	62K	71	4208	0.30	10	52K	43
	2	3948	0.30	17	121K	155	5092	0.40	17	102K	91
	5	4080	0.50	38	295K	407	5224	0.70	38	250K	235
	10	4276	0.90	73	586K	827	5388	1.10	73	497K	475
	20	4588	1.60	143	1.2M	1667	5644	1.80	143	993K	955
	50	5460	3.70	353	2.9M	4187	6344	4.20	353	2.5M	2395
	100	6892	7.40	703	5.7M	8387	7472	8.51	703	4.9M	4795

Table 4: Single Application Experiments

App.	No Pages	Manual Models Peak Mem Usage(KB)	Manual Models Running Time(s)	Manual Models Latency (s)	Manual Models Size Trace	Manual Models States Explored	Generated Models Peak Mem Usage(KB)	Generated Models Running Time(s)	Generated Models Latency (s)	Generated Models Size Trace	Generated Models States Explored
Direct	1 1	4328	0.40	11	111K	88	5792	0.50	11	67K	48
Copy	2 3	4752	0.50	23	251K	1132	6032	0.80	23	153K	419
&	7 10	6448	1.60	71	881K	10859	6632	3.20	71	538K	3324
Simple	14 20	9568	3.90	145	1.8M	38456	7852	10.51	145	1.1M	13176
Print	35 50	21048	14.01	367	4.4M	186191	13212	51.43	367	2.7M	69762
	70 100	49460	39.02	737	8.8M	600126	28524	180.01	737	5.4M	252272
	1400 1930	4177072	1090.46	-[1]	-[1]	-[1]	4165864	24405.98	-[1]	-[1]	-[1]
Direct	1 2	4552	0.60	15	174K	703	6640	0.90	15	107K	379
Copy&	5 10	6384	1.60	59	883K	10248	7524	5.00	59	577K	5180
Process	10 20	10040	4.34	114	1.8M	47503	9556	18.91	114	1.2M	22990
From	25 50	34908	21.21	279	4.4M	334468	22400	123.08	279	2.9M	156820
Store	50 100	123288	79.34	554	8.7M	1396743	66892	502.32	554	5.7M	647870
	450 900	4172460	1530.04	-[1]	-[1]	-[1]	-[1]	-[1]	-[1]	-[1]	-[1]

Table 5: Concurrent Application Experiments

[1]Out of memory

App.	No Pages	Manual Models					Generated Models				
		Peak Mem Usage(KB)	Running Time(s)	Latency (s)	Size Trace	States Explored	Peak Mem Usage(KB)	Running Time(s)	Latency (s)	Size Trace	States Explored
Simple Print	1	3924	0.20	7	26K	9	132	0.10	7	5.1K	5
	2	3936	0.20	11	49K	20	136	0.10	11	9.7K	10
	5	3984	0.30	23	118K	54	132	0.10	23	24K	24
	10	4044	0.40	43	233K	116	3260	0.20	43	46K	50
	20	4172	0.70	83	463K	264	3272	0.20	83	91K	98
	50	4516	1.50	203	1.2M	613	3316	0.30	203	229K	249
	100	5060	2.80	403	2.3M	1163	3388	0.60	403	460K	499
Process from Store	1	3228	0.20	9	44K	18	2792	0.20	9	15K	10
	2	3952	0.20	12	80K	68	2976	0.20	12	27K	31
	5	4024	0.40	24	211K	246	3812	0.30	24	71K	121
	10	4136	0.70	44	428K	656	3824	0.30	44	140K	281
	20	4348	1.20	84	863K	1476	3868	0.50	84	277K	601
	50	4956	2.73	204	2.2M	3936	3996	1.00	204	690K	1561
	100	5964	5.50	404	4.3M	8036	4212	1.90	404	1.4M	3161
Scan to Store	1	3632	0.20	9	35K	33	3516	0.20	9	28K	23
	2	3652	0.20	16	69K	69	4032	0.50	16	54K	45
	5	3696	0.30	37	168K	165	4220	0.40	37	132K	111
	10	3764	0.50	72	334K	325	4336	0.60	72	264K	221
	20	3904	0.90	142	667K	645	4352	0.80	142	527K	441
	50	4340	2.20	352	1.7M	1605	4604	2.20	352	1.3M	1101
	100	5056	4.20	702	3.3M	3205	5020	3.50	702	2.6M	2201
Direct Copy	1	3860	0.20	10	62K	71	5024	0.40	10	52K	43
	2	3892	0.30	17	121K	155	5048	0.40	17	102K	91
	5	3968	0.50	38	295K	407	5108	0.70	38	250K	235
	10	4076	0.90	73	586K	827	5200	1.10	73	497K	475
	20	4308	1.60	143	1.2M	1667	5388	1.90	143	993K	955
	50	5020	4.00	353	2.9M	4187	5952	4.20	353	2.5M	2395
	100	6188	7.30	703	5.7M	8387	6904	8.21	703	4.9M	4795

Table 6: Single Application Experiments with Progress Measures

App.	No Pages	Manual Models: Peak Mem Usage(KB)	Manual Models: Running Time(s)	Manual Models: Latency (s)	Manual Models: Size Trace	Manual Models: States Explored	Generated Models: Peak Mem Usage(KB)	Generated Models: Running Time(s)	Generated Models: Latency (s)	Generated Models: Size Trace	Generated Models: States Explored
Direct	1 1	4284	0.30	11	111K	88	5728	0.50	11	67K	48
Copy	2 3	4500	0.50	23	251K	1130	5876	0.80	23	153K	418
&	7 10	5480	1.60	71	881K	10578	6180	3.30	71	537K	3283
Simple	14 20	7400	3.60	145	1.8M	34858	6908	9.71	145	1.1M	11659
Print	35 50	12808	11.31	367	4.4M	149926	10040	39.72	367	2.7M	50857
	70 100	26568	27.92	737	8.8M	433816	19012	124.88	737	5.4M	161187
	1400 1930	4141568	1686.01	-[1]	98B	-[1]	3943984	36934.01	14622	107M	42906117
Direct	1 2	4456	0.40	15	174K	704	6484	0.80	15	107K	375
Copy&	5 10	5584	1.60	59	881K	10266	6860	4.90	59	576K	4884
Process	10 20	7540	4.10	114	1.8M	47551	7948	18.81	114	1.2M	21094
From	25 50	21352	19.51	279	4.4M	334606	15272	115.17	279	2.9M	142324
Store	50 100	71356	70.84	554	8.7M	1397031	41216	472.40	554	5.7M	586374
	450 900	4124876	2189.71	-[1]	-[1]	-[1]	-[2]	-[2]	4954	52M	48831174

Table 7: Concurrent Application Experiments with Progress Measures

[1] Out of memory
[2] No value specified by UPPAAL