# Model-Driven Design-Space Exploration for Embedded Systems: The Octopus Toolset*

Twan Basten[1,2], Emiel van Benthum[2], Marc Geilen[2], Martijn Hendriks[3], Fred Houben[3], Georgeta Igna[3], Frans Reckers[1], Sebastian de Smet[4], Lou Somers[2,4], Egbert Teeselink[2], Nikola Trčka[2], Frits Vaandrager[3], Jacques Verriet[1], Marc Voorhoeve[2], and Yang Yang[2]

[1]Embedded Systems Institute, [2]Eindhoven University of Technology, [3]Radboud University Nijmegen, [4]Océ Technologies
contact: a.a.basten@tue.nl (Twan Basten)

**Abstract.** The complexity of today's embedded systems and their development trajectories requires a systematic, model-driven design approach, supported by tooling wherever possible. Only then, development trajectories become manageable, with high-quality, cost-effective results. This paper introduces the Octopus Design-Space Exploration (DSE) toolset that aims to leverage existing modeling, analysis, and DSE tools to support model-driven DSE for embedded systems. The current toolset integrates Uppaal and CPN Tools, and is centered around the DSE Intermediate Representation (DSEIR) that is specifically designed to support DSE. The toolset architecture allows: (*i*) easy reuse of models between different tools, while providing model consistency, and the combined use of these tools in DSE; (*ii*) domain-specific abstractions to support different application domains and easy reuse of tools across domains.

**Key words:** Design-space exploration, Modeling, Analysis, Embedded Systems, CPN Tools, Uppaal

## 1 Introduction

High-tech systems ranging from smart phones to printers, from cars to radar systems, and from wafer steppers to medical imaging equipment contain an embedded electronic core that typically integrates a heterogeneous mix of hardware and software components. The resulting platform is often distributed, and it typically needs to support a mix of data-intensive computational tasks with event-processing control components. These embedded components more and more often have to operate in a dynamic and interactive environment. Moreover, not only functional correctness is important, but also quantitative properties related to timeliness, quality-of-service, resource usage, and energy consumption. The complexity of today's embedded systems and their development trajectories is thus increasing rapidly. At the same time, development trajectories are expected to produce high-quality and cost-effective products.

* This work was carried out as part of the Octopus project with Océ Technologies B.V. under the responsibility of the Embedded Systems Institute. This project is partially supported by the Netherlands Ministry of Economic Affairs under the Bsik program.

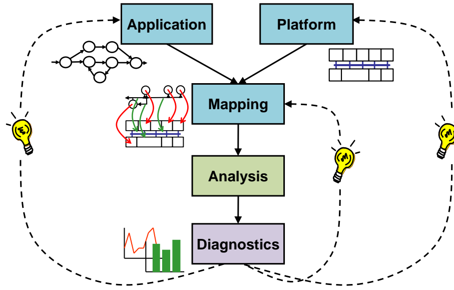



**Fig. 1.** The Y-chart DSE method.

A common challenge in development trajectories is the need to explore extremely large design spaces, involving multiple metrics of interest (timing, resource usage, energy usage, cost). The number of design parameters (number and type of processing cores, sizes and organization of memories, interconnect, scheduling and arbitration policies) is typically very large and the relation between parameter settings and design choices on the one hand and metrics of interest on the other hand is often difficult to determine. Given these observations, embedded-system design trajectories require a systematic approach, that is automated as far as possible. To achieve high-quality results, the design process and tooling needs to be model-driven. No single modeling approach or analysis tool is best fit for all the observed challenges. We propose to leverage the combined modeling and analysis power of various formal methods tools into one integrated Design-Space Exploration (DSE) framework, the *Octopus* framework. The framework is centered around an intermediate representation, *DSEIR* (Design-Space Exploration Intermediate Representation), to capture design alternatives. DSEIR models can be exported to various analysis tools. This facilitates reuse of models across tools and provides model consistency between analyses. The use of an intermediate representation supports domain-specific abstractions and reuse of tools across application domains.

The design of DSEIR follows the Y-chart philosophy [1, 16] that is popular in hardware design, see Fig. 1. The Y-chart philosophy is based on the observation that DSE typically involves the co-development of an application, a platform, and the mapping of the application onto the platform. Diagnostic information is used to, automatically or manually, improve application, platform, and/or mapping. DSEIR follows the Y-chart philosophy by supporting, and separating, the specification of its main ingredients, applications, platforms, and mappings. This separation is important to allow independent evaluation of various alternatives of one of these system aspects while fixing the others. Often, for example, various platform and mapping options are investigated for a fixed (set of) application(s).

This paper presents a first version of the Octopus DSE toolset. It integrates CPN Tools [13] for stochastic simulation of timed systems and Uppaal [2] for model checking and schedule optimization. Through an illustrative running example, we present DSEIR, its translations to CPN Tools and Uppaal, and the envisioned use of complementary formal analysis tools in a DSE process. We also evaluated DSEIR on two industrial printer data paths. Simulation and analysis

times in CPN Tools and Uppaal for automatically generated models are very similar to the simulation and analysis times for handcrafted models [12].

The paper is organized as follows. We discuss related work in Section 2. Section 3 introduces a typical DSE question, serving as our running example. Section 4 briefly presents the toolset architecture and current realization. DSEIR and the translations to CPN Tools and Uppaal are presented in Sections 5 and 6. Section 7 illustrates the use of the toolset for the running DSE example, and it summarizes the results for the two printer case studies. Section 8 concludes.

## 2 Related Work

There exists a plethora of academic and commercial frameworks supporting Y-chart-based DSE of embedded systems [4–7, 17, 22–25, 28]. Some support formal analysis, in particular the Metropolis/Metro II [5], SHE [25], and Uppaal-based [4, 6] frameworks. Others build on simulators, like SystemC or Simulink, and offer no or limited support for other types of analysis such as formal verification or scheduler/controler synthesis. Moreover, most frameworks provide their own modeling and analysis methods and do not support other input and output formats. In contrast, Octopus is open and extensible. Through its intermediate representation DSEIR, it provides a generic interface between input languages and analysis tools; its modeling features can be compared to those of an assembly language (albeit on a much higher abstraction level). Octopus intends to combine well established formal methods in one framework, so that every method can be used for what it is best for. Through its link with Uppaal, it is for example possible to integrate the schedulability analyses of [4, 6].

Closest to our work is the Metropolis/Metro II [5] framework. This also aims to be extensible, and to apply formal techniques in DSE. It provides several backends to interface with different analysis tools. The connection with the SPIN model checker in particular can be used to verify declarative modeling constraints expressed in linear temporal logic. The Octopus initiative is complementary to the Metro II framework in the sense that we have integrated different analysis tools. If Metro II is made available to the scientific community, then it should be straightforward to connect the two to combine their strengths.

The already discussed frameworks and tools are all based on the Y-chart approach. There is also a wide range of DSE tools and approaches for specific classes of systems that are not (explicitly) based on the Y-chart. These provide modeling and analysis support and/or automatically explore (parts of) a design space. A good overview of DSE tools can be found in [9]. More recent examples of academic tools are [15, 21]. Two industrial frameworks that provide formal modeling and analysis support for DSE are Scade [8] and SymTA/S [10].

Some other DSE frameworks target generic applicability while focusing on generic search and optimization algorithms (such as evolutionary algorithms) needed to explore the design space. Prominent examples are Pisa [3] and Opt4J [19]. These frameworks focus on automating the dashed feedback edges in Fig. 1. The current version of Octopus focuses on the analysis part of the Y-chart, i.e., obtaining the performance metrics for design alternatives. It is complementary to these other frameworks. As future work, we plan to connect Octopus to

frameworks like Pisa and Opt4J to automate the exploration of those parts of the design space that are amenable to automatic exploration.

## 3 Motivating Example

Fig. 2 presents a typical DSE problem. It is used as an illustrative example throughout the paper and as a test case for the Octopus framework. The example shows a pipeline of tasks for processing streams of data objects. The tasks are executed on a heterogeneous multiprocessor platform with two memories and a bus-based interconnect. The example system incorporates ingredients typically observed in today's embedded systems. The task pipeline exhibits dynamic workload variations, depending on the complexity of the data object being processed. Video decoding and encoding, graphics pipelines, and pdf processing for printing are applications showing such data-dependent workload variations. The example platform combines a general-purpose processor (CPU), a graphics processor (GPU) and an FPGA (Field-Programmable Gate Array). Such a mix is often chosen for performance and energy efficiency. An FPGA for example is well suited for executing regular data-intensive computation kernels without much dynamic variation. Multiple kernels can share an FPGA and execute in parallel. A typical DSE question for a system as illustrated in Fig. 2 is to optimize performance while minimizing the resources needed to obtain the performance.

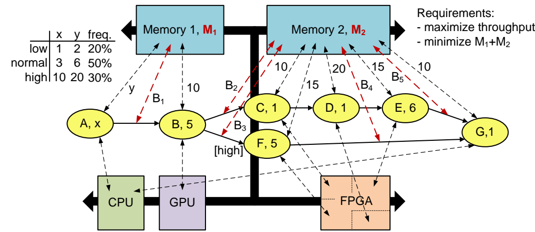



**Fig. 2.** A running example.

The detailed specification of the example system is as follows. The application has tasks $A$ through $G$, mapped onto a platform consisting of one CPU, one GPU, an FPGA, and two memories, all connected by a bus. Tasks process data objects, one at a time, and need to be executed in an iterative, streaming fashion. The DSE question is to *minimize the memory sizes* $M_1$ and $M_2$ such that *throughput* in terms of data objects per time unit is *maximized*, and to find *a simple scheduling policy that achieves this throughput.*

Tasks $A$ and $G$ share the CPU. The order of execution can be chosen freely and preemption is possible. Task $B$ runs on the GPU. Tasks $C$ through $F$ share the FPGA but can be executed in parallel. The annotated dashed arrows between tasks and memories specify memory requirements. The indicated amount is claimed at the task execution start and released at the task completion. The numbers inside tasks indicate the execution time of one execution of the task. A task can only be executed if all required memory is available.

Task $A$ has three execution modes with different workloads (low, normal, high, with average occurrence frequencies of 20%, 50%, and 30%, resp.). The different workloads come with different execution times and memory requirements as indicated in the small table in the figure. Task $F$ is only executed for objects with high workloads, indicated in the figure by the '[high]' precondition.

The edges between tasks indicate data dependencies. Five dependencies involve the transfer of data objects between processing resources. These transfers use the memories, as indicated by the dashed arrows $B_1$ through $B_5$. A data object corresponds to 10 memory units. The memory for an object to be produced by a task execution on an output edge is needed at the execution start, to make sure that sufficient space is available. The data object is available for the successor task immediately upon completion of the producing task. The memory of an object being read by a task is released at the execution end, so that the complete data object is available during the entire task execution. The data transfers between tasks $C$, $D$, and $E$ are handled inside the FPGA.

For simplicity, we assume that context switching time (on the CPU) and time needed for data transfers over the bus are included in the task execution time.

We would like to leverage existing formal analysis tools to solve the sketched DSE problem. The system combines a stochastic part, the $A$-$B$ subsystem, and a relatively static part, the $B$-$G$ subsystem. Simulation tools like CPN Tools are particularly suitable to analyze stochastic (sub)systems, whereas model checking tools like Uppaal are well suited to optimize non-stochastic (sub)systems.

## 4 The Octopus Architecture and Current Realization

The Octopus toolset aims to be a (*i*) *flexible toolset* that provides (*ii*) *formal analysis support* for DSE by (*iii*) leveraging *existing tools*, also *in combination*; the toolset should be applicable (*iv*) *across application domains* within the embedded-systems domain, and it should allow (*v*) *domain-specific modeling abstractions*. To achieve these goals, the most important architectural choice, illustrated in Fig. 3, is to separate the main ingredients of the Y-chart approach into separate components with well-defined interfaces, centered around an intermediate representation, DSEIR, specifically designed for Y-chart-based DSE. We distinguish three groups of components: (*i*) editing support, for applications, platforms and mapping; (*ii*) analysis support; (*iii*) diagnostics and visualization.

The design of DSEIR is crucial in achieving our goals. It should be sufficiently expressive to allow the specification of design alternatives at the required level of detail. At the same time, it should be well structured and precisely defined, to allow model transformations to various analysis tools that preserve properties of interest and that provide models that can be efficiently analyzed in the targeted tools. The next two sections explain DSEIR and the transformations to CPN Tools and Uppaal that currently have been implemented. Section 7 provides experimental results that indicate that DSEIR achieves its goals.

The right part of Fig. 3 shows the current Octopus implementation. At the core is a Java library implementing DSEIR. Modules DSEIR2CPN and DSEIR2Uppaal implement the interfaces with CPN Tools and Uppaal. A domain-specific Data Path Editor for modeling printer data paths has been developed

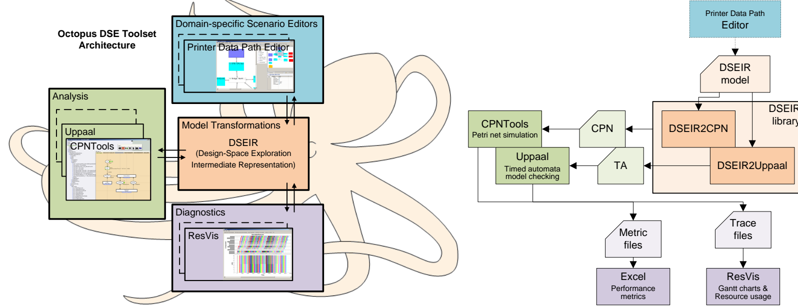



**Fig. 3.** The Octopus toolset: Architecture (left) and current realization (right).

and will be integrated in the near future. A printer data path is the electronic heart of a printer that does the image processing for printing, scanning, and copying. Analysis output is visualized through Excel, for (trends in) typical performance metrics such as throughput and memory usage, and ResVis, a simple but powerful tool for visualizing Gantt charts and resource usage over time.

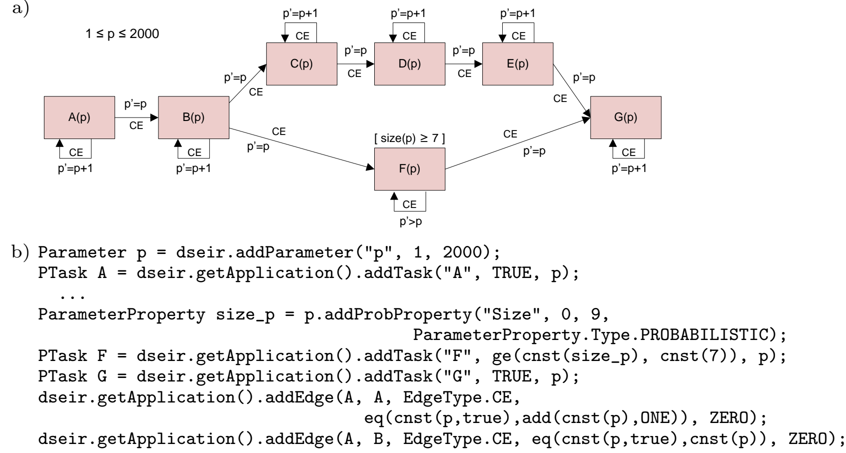
## 5 DSEIR

DSEIR is implemented as a Java library. Specifications can be entered directly in Java or in an XML format derived from the Java implementation. We present the main principles of DSEIR, focusing on the Java interface. The formal definition of DSEIR and its operational semantics can be found in [26].

**Application modeling:** DSEIR captures the application side of the Y-chart by means of *Parameterized Timed Partial Orders* (PTPOs). A PTPO is a standard task graph, extended with (*i*) *task parameters* – for simplifying large (and even infinite), albeit structured, precedence relations, (*ii*) *time* – for specifying minimal timing delays between tasks, and (*iii*) four different *types* of precedence rules, for fine-grained specifications in terms of task *enabling* and *completion* events. The expressivity of PTPOs is that of timed event-based partial orders; every PTPO can, in principle, be unfolded to such a, possibly infinite, structure.

Fig. 4a shows the PTPO for our example. Tasks have a parameter $p$, identifying the objects being processed. Its range is specified in the top left. The condition above $F$ restricts the scope of $p$ for $F$ to objects with high workloads (with size 7–9). A condition $p' = p$ on an edge denotes that the precedence only exists between the same instance of the left and right tasks (the $p'$ is the $p$ of the right task). The PTPO thus specifies that $A(1)$ should be executed before $B(1)$, $B(1)$ before $C(1)$, etc. There is no direct causal relationship between e.g. $A(2)$ and $C(1)$. The self-loop precedences with condition $p' = p + 1$ eliminate auto-concurrency, i.e., they ensure that no two instances of the same task can be executed at the same time. The self-loop for task $F$ requires condition $p' > p$ because, due to the conditional execution, the range of objects it processes may be non-consecutive. We need only the CE precedence type, specifying a causal relation between the completion (C) of one task and the enabling (E) of another. The other three E-C combinations are typically used for specifying task periods

a)




b)
```
Parameter p = dseir.addParameter("p", 1, 2000);
PTask A = dseir.getApplication().addTask("A", TRUE, p);
  ...
ParameterProperty size_p = p.addProbProperty("Size", 0, 9,
                                    ParameterProperty.Type.PROBABILISTIC);
PTask F = dseir.getApplication().addTask("F", ge(cnst(size_p), cnst(7)), p);
PTask G = dseir.getApplication().addTask("G", TRUE, p);
dseir.getApplication().addEdge(A, A, EdgeType.CE,
                               eq(cnst(p,true),add(cnst(p),ONE)), ZERO);
dseir.getApplication().addEdge(A, B, EdgeType.CE, eq(cnst(p,true),cnst(p)), ZERO);
```

**Fig. 4.** a) PTPO for the running example; b) part of its Java DSEIR specification.

and minimal durations. The example PTPO has no time constraints (all minimal delays zero); we do not impose any delays at the application level, but expect them at the resource level. Fig. 4b shows how the PTPO is specified in DSEIR.

**Specifying a platform:** A platform in DSEIR is a set of generic *resources*. Each resource has a *name* and a *capacity* (an integer at least zero), and can support preemption or not. No distinction is made between computational, storage and communications resources, nor is any connection between resources explicitly specified. Fig. 5a shows the specification of the CPU and the two memories. Both memories have a capacity of 200 units; the CPU has capacity 1, modeling that it is either free or busy. The preemption flag is true only for the CPU.

a)
```
Resource cpu = dseir.getPlatform().addResource("cpu", 1, true);
Resource mem1 = dseir.getPlatform().addResource("mem1", 200, false);
Resource mem2 = dseir.getPlatform().addResource("mem2", 200, false);
```
b)
```
dseir.getMapping().addDuration(A, iF(lt(cnst(size_p),cnst(2)),cnst(1), iF(
                               lt(cnst(size_p),cnst(7)),cnst(3),cnst(10))));
dseir.getMapping().addUtilizationBound(A, cpu, ONE, ONE);
dseir.getMapping().addUtilizationBound(A, mem1, iF(lt(cnst(size_p),cnst(2)),cnst(12), iF(
      lt(cnst(size_p),cnst(7)),cnst(16),cnst(30))), iF(lt(cnst(size_p),cnst(2)),cnst(12),
                  iF(lt(cnst(size_p),cnst(7)),cnst(16),cnst(30))));
```
c)
```
dseir.getMapping().addHandover(BC, mem2, cnst(10));
dseir.getMapping().addHandover(BF, mem2, iF(lt(cnst(size_p),cnst(7)),cnst(0),cnst(10)));
```
d)
```
dseir.getScheduler().addPriority(A, sub(cnst(1), mult(cnst(10), cnst(p))));
dseir.getScheduler().addPriority(B, sub(cnst(2), mult(cnst(10), cnst(p))));
  ...
dseir.getScheduler().setPreemptive(A, cpu, TRUE);
```

**Fig. 5.** The running example in DSEIR a) resources; b) duration function; c) handover; d) scheduling.

**Mappings:** The mapping part of the Y-chart is captured in DSEIR through the concepts of a *duration function* and a *resource handover function*.

A duration function specifies the duration of a task for any possible resource configuration. If the duration is zero, a task execution is instantaneous once it

gets its resources; if the duration is infinite, a task makes no progress with the given resources. The specification of a duration function is split into the specification of the *minimum-required* resource configuration (the unique minimal configuration yielding finite duration), the maximum-required resource configuration (the unique minimal configuration yielding the shortest possible duration) and the actual duration for configurations in between. This can be seen as specifying interval-based resource claims for every task, quantifying resource sharing.

Computational-resource claims that allow sharing are usually specified with the minimum-required configuration of 1 (at least one resource unit is needed), maximum-required configuration of 100 (with less than 100 resource units the task does not run at full pace) and a linear function mapping assigning duration for every capacity in the interval $[1, 100]$. If a task needs exclusive access to a certain amount of resource, as the tasks in our example, then both the minimum-required configuration and the maximum-required one should be equal to the required amount. Storage resources are typically claimed in this way.

Fig. 5b shows the duration function for task $A$. The first line defines how the duration depends on the parameter of $A$. The expression has no resource information as $A$ only claims fixed resource amounts. The next lines specify that $A$ needs the CPU exclusively, and that it claims 12, 16 or 30 units of Memory 1 (including the 10 units for output), depending on the current object's size.

Often, a task reads, and subsequently deletes, data that some previous task produced. It is, moreover, sometimes desired to have a task reserve some resource for an upcoming task. DSEIR captures these situations by means of *handover functions*. A handover function for a task specifies the amount of resources that are kept upon its completion for some other task. Handovers are typically assigned to CE edges. Fig. 5c specifies the memory sharing between task $B$ and tasks $C$ and $F$. The two lines describe that $B$ leaves 10 units of Memory 2 to both $C$ and $F$, but the latter only when $F$ is to be executed for the same object.

**Scheduling:** Schedulers are part of the platform in the Y-chart; concrete policies for specific resources and applications are part of the mapping. Nevertheless, DSEIR treats scheduling separately. We predefined a *preemptive*, *event-driven*, *priority-based* scheduler that is activated each time a task is enabled or finishes. The user can specify priorities and the tasks that allow their resources to be taken away at run time (for resources that allow preemption). Fig. 5d shows an example priority assignment for tasks $A$ and $B$ that gives a task processing a lower-numbered object priority over a task processing a higher-numbered one. The code also specifies that task $A$ may be preempted.

**Extensions:** The current version of DSEIR supports the three main ingredients of the Y-chart approach. Future extensions will provide support for the DSE process itself, allowing the user to specify experiments, properties and performance metrics of interest, and verification and optimization objectives. We also consider developing a language to support the compositional specification of schedulers (e.g., in the style of [18, 20]). Complex multiprocessor platforms typically contain multiple interacting schedulers (including arbiters for e.g. buses and memories). Structured language support for scheduling is an enabler for formal analysis of such compositions of schedulers.

## 6 Model Transformations

The current toolset has interfaces to CPN Tools and Uppaal. This section introduces the model transformations implemented to translate DSEIR models to Coloured Petri Nets (CPNs) and Timed Automata (TA). Based on the semantics of CPN [14], Uppaal TA [27], and DSEIR [26], it can be shown that both translations preserve equivalence; the models generate observation equivalent [11] timed transition systems when considering task enabling and completion events.

### 6.1 Transforming DSEIR Models to Coloured Petri Nets

CPNs [14] are a well-established graphical formalism, extending classical Petri nets with data, time, and hierarchy. CPNs have been used in many domains (manufacturing, workflow management, distributed computing, embedded systems) and are supported by CPN Tools [13], a powerful framework for modeling, functional verification (several techniques) and performance analysis (stochastic discrete-event simulation). This makes CPNs very suitable for Octopus.

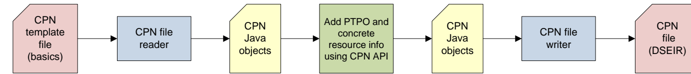



**Fig. 6.** Translating DSEIR specifications to CPN models

The interface between the DSEIR library and CPN Tools is realized in the DSEIR2CPN module. Any DSEIR specification can be converted. Several special cases are recognized to optimize the translated model in terms of simulation time. For maintainability and extensibility, the conversion uses a CPN template file containing the basic structure of the CPN model, high-level dynamics of the resource handler and monitors for producing simulation output. The information from DSEIR is added to this template, generating an executable CPN model. To allow reading and writing CPN models in Octopus, we built a Java API for a reasonably large class of CPN models. Fig. 6 shows the conversion process.

Fig. 7 shows the generated CPN model for our example, with a manually improved graphical layout to improve presentation. We do not go into details of the transformation, but rather briefly explain the generated model. The SYS CPN page shows that, at the highest level, the system consists of a PTPO (page TPO), a resource handler (page RH) and the interface between them (places `TPO_to_RH` and `RH_to_TPO`). In the picture only the translations of tasks $E$ and $F$ and their connections to $B$, $D$, and $G$ are shown. Tasks are split into an *enabling* and *completion* event, modeled as transitions with a place in between (`E_e`, `E_c`, and `p_EC_E` for task $E$); these transitions are connected to the appropriate interface places `to_RH` and `from_RH`. Parameter-scope restrictions are modeled as guards (only task $F$ has a guard – predicate `Size`). Precedences are captured by a place between the two corresponding transitions of these tasks (place `p_D_c_E_e` for the CE precedence between $D$ and $E$). A special `init` transition (not in the picture) is always the first transition to fire in a model. Its main purpose is to take a sample for any stochastic property in the PTPO (the workload in our

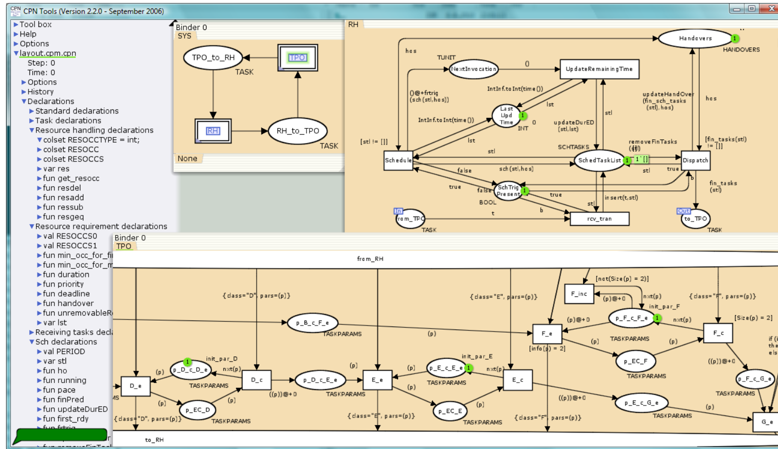



**Fig. 7.** CPN model obtained from the DSEIR specification of the running example

example). The TPO page is the only part of the CPN model that has to be completely generated from DSEIR. The RH page is to a large extent generic for every DSEIR model, with many parts predefined in the template. Most of the scheduling dynamics is hidden in the CPN ML functions on the left.

### 6.2 Transforming DSEIR Models to Timed Automata

Timed Automata (TA) are a suitable formalism to capture at least a subset of the systems that DSEIR can express. Powerful tools, the model checker Uppaal [2] being one of them, are available for the analysis of TA. Uppaal's powerful analysis engine and relatively mature modeling support (ease of use, extensiveness of the modeling language) make it very suitable for integration in Octopus.

The DSEIR2Uppaal module implements the translation of DSEIR to Uppaal's input language. It unfolds the PTPO to a concrete task partial order without parameters, which is possible for finite parameter ranges. Each task instance is modeled by a separate TA. The TA broadcast their start and end events via *channels*. Each TA has a number of *waiting* locations, and the *enabled*, *active* and *done* locations. Location *enabled* indicates that a task is schedulable for execution. Fig. 8 shows the TA for the first instance of task $G$ of our example. It has two *waiting* locations, as $G$ must wait for tasks $E$ and $F$. When $G$ is enabled, it can actually start if it can take all required resources, which is modeled by the guard $canClaim()$. If it starts, it broadcasts its start over the $started0$ channel, takes the resources via the $claim()$ function, and resets its $x_cpu$ clock. The transition from *active* to *done* then happens after exactly $time()$ time units. The TA broadcasts that it is finished, and releases the resources.

A resource that is not shared is modeled by an integer variable that represents the remaining capacity and one or more clocks. These resources are claimed by decreasing the variable (function $claim()$ in Fig. 8), and released by increasing the variable (function $release()$). If a task TA has claimed a resource, then it can use the clock of that resource for its timing. The TA in Fig. 8 uses one of

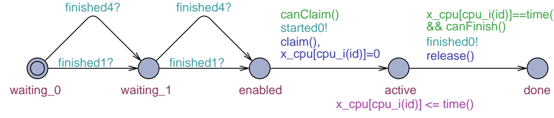



**Fig. 8.** The Uppaal timed automaton specification for the first instance of task $G$.

the *x_cpu* clocks. Shared resources are modeled by a set of TA that implement preemption and redistribution of the resource amounts when a task starts or stops using the resource. Preemption cannot be modeled exactly with Uppaal, but it can be approximated with arbitrary precision [12]. Unfortunately, this technique fragments the symbolic representation of time that Uppaal uses and will generally have a negative effect on the performance of the analysis.

As in the CPN case, special cases of the transformation may be recognized. If no two instances of the same task can run concurrently, it is not necessary to unfold the PTPO. All instances of one task can then be captured in a single, iterative TA. This limits the number of automata in the model, thereby improving performance and reducing the size (in terms of bytes of memory) of a state.

## 7 Case Studies

Our first experimental evaluations with the Octopus toolset involve the running example used throughout this paper and two printer data path designs.

### 7.1 The Running Example

We use the combined strengths of CPN Tools and Uppaal, i.e., (stochastic) simulation and schedule optimization, to solve the DSE problem of Section 3.

**Problem refinement:** From the expected CPU workload of 5.7 per data object and the task $E$ execution time of 6 time units, we conclude that 1/6 objects/time unit is an upper bound for throughput. The $B$-$G$ subsystem allows this throughput if memory $M_2$ is sufficiently large. Due to incidental peak workloads on the CPU, this bound however can never be reached. It can be approached arbitrarily close though with a sufficiently large $M_1$ memory. We decide to aim for memory sizes that allow a throughput within 2% of the upper bound. We also want a simple scheduling policy that provides this throughput. Because of the stochastic nature of the system, we are satisfied if repeated simulations show that the desired throughput is achieved with a 99% confidence.

**Initial evaluation:** The DSEIR model used throughout Section 5 (without task priorities, because the specification does not give priorities) is our initial model. Memories $M_1$ and $M_2$ are both set to the sufficiently large value of 200. Input streams, sampled from the distribution in Fig. 2, have 2000 objects.

Initial CPN Tools simulations show deadlocks. These are caused by the memory allocation strategy, and occur both on $M_1$ and on $M_2$. Both the $A$-$B$ and the $B$-$G$ subsystems are pipelines. If the heads of those pipelines work ahead too far and claim all the memory, tasks further down the pipeline are blocked.

The deadlocks occur independent of the memory sizes. We therefore change the memory allocation, switching from task-based allocation to object-based allocation. In this strategy, the first task needing a memory resource ($A$ for $M_1$, $B$

for $M_2$) claims the maximum amount needed in the pipeline and upon termination hands over to its successor the maximum amount needed in the remaining pipeline, releasing the rest (if any). This handover and release strategy is adopted by all tasks in the pipeline. This strategy is deadlock-free by construction.

$M_2$ **optimization:** The next steps further investigate the $B$-$G$ subsystem, to optimize $M_2$. At the end, we then plan to reduce $M_1$ as far as possible. We start with a binary search using CPN Tools simulations to determine an initial $M_2$ bound with the object-based memory allocation. We further prioritize the execution of tasks further down the pipeline over tasks earlier in the pipeline. This is not necessary but it is expected to give better performance. It turns out that an $M_2$ size of 120 is needed for the desired throughput.

We then want to use Uppaal to investigate whether it is possible to optimize the $B$-$G$ subsystem and further reduce the $M_2$ size by smart scheduling. For a sufficiently large $M_1$, we may assume that $B$ always has data to execute. Removing task $A$ and memory $M_1$ and conservatively (from the resource usage perspective) assuming that $F$ is always executed then removes all stochastic behavior in the model, allowing the use of Uppaal. To maximize scheduling freedom, we remove task priorities and revert back to task-based memory allocation (which is more efficient than object-based allocation). We use an observer TA to monitor throughput. A binary search for $M_2$ with the upper bound from the simulations shows that a size of 110 is needed to allow an execution with the optimal throughput (which is 1/6 objects/time unit for this subsystem).

Given that a throughput-optimal execution exists for an $M_2$ size smaller than the earlier bound of 120, we want to find simple scheduling rules that provide an optimal throughput within this 110 bound. Not all executions are throughput-optimal. Some, for example, still lead to deadlock; as before, tasks early in the pipeline work too far ahead of tasks later in the pipeline. We decide to investigate scheduling rules $XY(k)$ that constrain the difference between the execution counts of tasks $Y$ and $X$ by integer $k$. For example, $GB(3)$ states that $B(p+3)$ should not become enabled before $G(p)$ has completed. These rules can be integrated into the PTPO via CE precedences. We investigate rules $BG(k)$, which limit how far tasks early in the pipeline can work ahead. We further investigate rule $DF(0)$, which guarantees that $F$ becomes only enabled after completion of $D$ on the same object. This disallows the simultaneous execution of memory-expensive tasks $D$ and $F$, exploiting the fact that $F$ is not in the time-critical path of the application. Inspection of optimal schedules obtained with Uppaal suggests that rule $GF(2)$ would be a possible alternative for $DF(0)$. $GF(2)$ has a similar effect but is less restrictive.

We combine the scheduling rules with *greedy scheduling*, without priorities, that non-deterministically resolves conflicting memory claims. It turns out that $GB(3)$ with $GF(2)$ is the best combination. With an $M_2$ size of 110, it *guarantees* an optimal throughput for all executions. $GB(4)$ and $GB(5)$, both with $GF(2)$, provide good alternatives, guaranteeing an optimal throughput when $M_2$ is 120 and 130, respectively. When using the *task priorities* introduced earlier for simulations both combinations give optimal throughput also with an $M_2$ size of 110. These alternatives are of interest because they are less sensitive to intermittent stalls of task $B$, which may occur in the full system due to peak workloads

for task $A$. $GF(2)$, which we would not have considered without analyzing the Uppaal results, outperforms $DF(0)$.

**Final optimization:** For the final optimization step, we take the full system with object-based memory allocation for the $A$-$B$ subsystem and task-based memory allocation for the $B$-$G$ subsystem. We investigate the three mentioned combinations of scheduling rules, with greedy, priority-based scheduling (with the mentioned priority scheme). The full system should achieve a throughput of at most 2% below the bound of 1/6 objects/time unit. CPN Tools simulations indicate that the $GB(5)$-$GF(2)$ combination performs best. An $M_1$ size of 110 suffices, and due to the slightly relaxed throughput constraint and the fact that $F$ is not always executed, the size of $M_2$ can be further reduced to 100. Thus, the final result of the DSE is that $M_1$ and $M_2$ sizes need to be 110 and 100.

**Concluding remarks:** The unique selling point of a model checker is its ability to exhaustively explore the state space of a model and check whether desired properties hold. In this case study, Uppaal has been used to prove that *any greedy scheduler* in combination with various scheduling rules and memory sizes *guarantees* the required throughput. Such results are hard if not impossible to obtain with simulators. On the other hand, Uppaal cannot handle stochastic behavior, and it does not scale to very large models. With limited input streams for the Uppaal analyses, all simulation and analysis experiments performed for this DSE take at most a few minutes on normal laptops. The example DSE illustrates that CPN Tools and Uppaal may complement each other. The automatic translations to CPN Tools and Uppaal from common DSEIR specifications made experimenting very easy.

### 7.2 Modeling Printer Data Paths

To evaluate the expressive power of DSEIR, we did two case studies involving digital printer data paths of Océ Technologies, based in Venlo, the Netherlands.

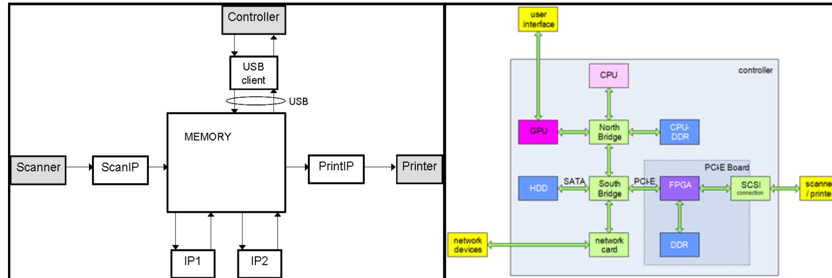



**Fig. 9.** The platforms in the printer case studies.

The left picture in Fig. 9 is an abstracted version of an FPGA-based platform. It is used in a printer that supports use cases such as printing, copying, scanning, scan-to-email, and print-from-store. The machine can be accessed locally, through the scanner and the controller, and remotely, through the controller. The use cases all use different components in the platform. Various tasks can execute in parallel. Resources like the memory and the associated memory bus, as well

as the USB are shared among tasks and among print and scan jobs running in parallel. The available USB bandwidth moreover dynamically fluctuates depending on whether it is used in one or in two directions simultaneously. Finding the fastest schedule for a sequence of jobs on the sketched platform is non-trivial. In [12], we modeled and analyzed this system with Uppaal. A novelty introduced in [12] was the already mentioned discrete approximation of the dynamic USB behavior. For the current paper, we modeled the system in DSEIR, automatically generated Uppaal models, and compared the results with the results obtained from the handcrafted models used for [12]. We achieved the same results, with similar analysis times (typically in the order of a few minutes).

The right picture in Fig. 9 shows a heterogeneous multiprocessor platform that combines one or more CPUs (running windows) with a GPU, one or more Harddisks (HDDs), and an FPGA. Because of heterogeneity and the use of general CPUs, the platform is more challenging than the platform of the first case study. We modeled the print use case for this platform in CPN Tools, to analyse the achieved throughput in images per second under dynamically fluctuating workloads, and to find out the appropriate buffer sizes between components. We later modeled the print use case on this platform in DSEIR and automatically derived a CPN model. Also in this case, we obtained matching results.

Together, the two case studies show that DSEIR is sufficiently expressive to capture a variety of realistic systems.

## 8 Conclusions

We have presented our ideas about model-driven design-space exploration (DSE) for embedded systems, and a first version of the Octopus DSE toolset based on these ideas. This toolset is organized around an intermediate representation, DSEIR, that is specifically designed to support DSE. It allows the independent specification of applications, platforms, and mappings in a compact and precise way. The notion of a Parameterized Timed Partial Order for capturing applications is a novel element of DSEIR. The toolset is open and extensible. It aims to integrate existing formal analysis and simulation tools in the DSE process, to leverage their combined strengths. Our case studies show that combining tools is meaningful and useful. The toolset, DSEIR in particular, provides model consistency between analyses with different tools and easy experimentation. It allows easy reuse of tools among application domains by providing an intermediate between domain-specific modeling abstractions and formal analysis tools.

In future work, we plan to evaluate the toolset in other application domains. The current work mostly focuses on connecting tools, and on modeling and analyzing design alternatives. Future work will also focus on capturing experiment design and on automating (parts of) the DSE process itself, among others schedule optimization. We plan to develop a structured DSE method in which tools complement each other. New analysis tools will be added when the need or opportunity arises. Other relevant additions are a model repository with support for model versioning, and decision support. An interesting extension beyond the core DSE process is code generation support. The latest information about the Octopus toolset is available through `dse.esi.nl`.

## References


1. F. Balarin et al. *Hardware-Software Co-design of Embedded Systems: The POLIS Approach.* Kluwer, 1997.
2. G. Behrmann et al. Uppaal 4.0. *Proc. QEST*, p. 125–126. IEEE, 2006.
3. S. Bleuler et al. PISA – A Platform and Programming Language Independent Interface for Search Algorithms. *Proc. EMO*, LNCS 2632, p. 494–508. Springer, 2003.
4. A.W. Brekling et al.. Models and Formal Verification of Multiprocessor System-on-Chips. *J. Log. Algebr. Program.*, 77(1–2):1–19, 2008.
5. A. Davare et al. A Next-Generation Design Framework for Platform-based Design. *Proc. DVCon 2007*, February 2007.
6. A. David et al. Model-based Framework for Schedulability Analysis Using Uppaal 4.1. *Model-based Design for Embedded Systems*, p. 121–143. Taylor & Francis, 2009.
7. CoFluent Design. CoFluent Studio. www.cofluentdesign.com, 2010.
8. Esterel, Scade. www.esterel-technologies.com/products/scade-suite, 2010.
9. M. Gries. Methods for Evaluating and Covering the Design Space during Early Design Development. *Integration, the VLSI Journal*, 38:131–183, 2004.
10. A. Hamann et al. A Framework for Modular Analysis and Exploration of Heterogeneous Embedded Systems. *Real-Time Systems*, 33:101–137, 2006.
11. M. Hennessy and R. Milner. On Observing Nondeterminism and Concurrency. In *Proc. ICALP 1980*, LNCS 85, p. 299–309. Springer, 1980.
12. G. Igna et al. Formal Modeling and Scheduling of Data Paths of Digital Document Printers. *Proc. FORMATS 2008*, LNCS 5215, p. 170–187. Springer, 2008.
13. K. Jensen et al. Coloured Petri Nets and CPN Tools for Modelling and Validation of Concurrent Systems. *STTT*, 9(3–4), 2007.
14. K. Jensen and L.M. Kristensen. *Coloured Petri Nets.* Springer, 2009.
15. J. Keinert et al. SystemCoDesigner – An Automatic ESL Synthesis Approach by Design Space Exploration and Behavioral Synthesis for Streaming Applications. *ACM Trans. Design Automation of Electronic Systems*, 14: Art. No. 1, 2009.
16. B. Kienhuis et al. An Approach for Quantitative Analysis of Application-specific Dataflow Architectures. In *Proc. ASAP 1997*, p. 338–349. IEEE, 1997.
17. A. Ledeczi et al. Modeling Methodology for Integrated Simulation of Embedded Systems. *ACM Trans. Model. Comput. Simul.*, 13(1):82–103, 2003.
18. I. Lee et al. Resources in Process Algebra. *J. Log. Algebr. Progr.*, 72:98–122, 2007.
19. M. Lukasiewycz et al. Opt4J: Meta-heuristic Optimization Framework for Java. opt4j.sourceforge.net, 2010.
20. M.R. Mousavi et al. PARS: A Process Algebra with Resources and Schedulers. *Proc. FORMATS 2003*, LNCS 2791, p. 134–150. Springer, 2004.
21. G. Palermo et al. Multi-objective Design Space Exploration of Embedded Systems. *J. Embedded Computing*, 1:305–316, 2005.
22. A.D. Pimentel. The Artemis Workbench for System-Level Performance Evaluation of Embedded Systems. *Intl J. Embedded Systems*, 3(3):181–196, 2008.
23. I. Sander and A. Jantsch. System Modeling and Transformational Design Refinement in ForSyDe. *IEEE T. Comput.-Aid. Design*, 23(1):17–32, 2004.
24. MLDesign Technologies. MLDesigner. www.mldesigner.com, 2010.
25. B.D. Theelen et al. Software/Hardware Engineering with the Parallel Object-Oriented Specification Language. *Proc. Memocode 2007*, p. 139–148. IEEE, 2007.
26. N. Trcka et al. Parameterized Timed Partial Orders with Resources: Formal Definition and Semantics. Tech. Rep. ESR-2010-01, Eindhoven Univ. of Tech., 2010.
27. Uppaal Web Site. www.uppaal.com, Web Help, 2010.
28. I. Viskic et al. Design Exploration and Automatic Generation of MPSoC Platform TLMs from Kahn Process Network Applications. *Proc. LCTES*, p. 77–84. ACM, 2010.