Chapter 7
Model-Driven Design-Space Exploration for Software-Intensive Embedded Systems

Twan Basten, Martijn Hendriks, Nikola Trčka, Lou Somers, Marc Geilen, Yang Yang, Georgeta Igna, Sebastian de Smet, Marc Voorhoeve[†], Wil van der Aalst, Henk Corporaal, and Frits Vaandrager

Twan Basten
Embedded System Institute, P.O. Box 513, 5600 MB Eindhoven, The Netherlands &
Eindhoven University of Technology, Faculty of Electrical Engineering, Electronic Systems group, P.O. Box 513, 5600 MB Eindhoven, The Netherlands
e-mail: a.a.basten@tue.nl

Martijn Hendriks
Embedded Systems Institute, P.O. Box 513, 5600 MB Eindhoven, The Netherlands
e-mail: martijn.hendriks@esi.nl

Nikola Trčka
United Technologies Research Center, 411 Silver Lane, East Hartford, CT 06108, United States
Nikola Trčka was employed at Eindhoven University of Technology when this work was done.

Lou Somers
Océ-Technologies B.V., P.O. Box 101, 5900 MA Venlo, The Netherlands &
Eindhoven University of Technology, Faculty of Mathematics and Computer Science, Software Engineering and Technology group, P.O. Box 513, 5600 MB Eindhoven, The Netherlands
e-mail: lou.somers@oce.com

Marc Geilen · Yang Yang · Henk Corporaal
Eindhoven University of Technology, Faculty of Electrical Engineering, Electronic Systems group, P.O. Box 513, 5600 MB Eindhoven, The Netherlands
e-mail: {m.c.w.geilen,y.yang,h.corporaal}@tue.nl

Georgeta Igna · Frits Vaandrager
Radboud University Nijmegen, Institute for Computing and Information Sciences, Department of Model-Based System Development, P.O. Box 9010, 6500 GL Nijmegen, The Netherlands
e-mail: {g.igna,f.vaandrager}@cs.run.nl

Sebastian de Smet
Océ-Technologies B.V., P.O. Box 101, 5900 MA Venlo, The Netherlands
e-mail: sebastian.desmet@oce.com

[†]Marc Voorhoeve
Eindhoven University of Technology, Faculty of Mathematics and Computer Science, Architecture of Information Systems group, P.O. Box 513, 5600 MB Eindhoven, The Netherlands
5 April 1950 - 7 October 2011

Wil van der Aalst
Eindhoven University of Technology, Faculty of Mathematics and Computer Science, Architecture of Information Systems group, P.O. Box 513, 5600 MB Eindhoven, The Netherlands
e-mail: w.m.p.v.d.aalst@tue.nl

Abstract The complexity of today's embedded systems is increasing rapidly. Ever more functionality is realised in software, for reasons of cost and flexibility. This leads to many implementation alternatives that vary in functionality, performance, hardware, etc. To cope with this complexity, systematic development support during the early phases of design is needed. Model-driven development provides this support. It bridges the gap between ad-hoc back-of-the-envelope or spreadsheet calculations and physical prototypes. Models provide insight in system-level performance characteristics of potential implementation options and are a good means of documentation and communication. They ultimately lead to shorter, more predictable development times and better controlled product quality. This chapter presents the Octopus tool set for model-driven design-space exploration. It supports designers in modelling and analysing design alternatives for embedded software and hardware. It follows the Y-chart paradigm, which advocates a separation between application software functionality, platform implementation choices, and the mapping of software functionality onto the platform. The tool set enables fast and accurate exploration of design alternatives for software-intensive embedded systems.

7.1 Motivation

Industries in the high-tech embedded systems domain (including for example professional printing, lithographic systems, medical imaging, and automotive) are facing the challenge of rapidly increasing complexity of next generations of their systems: Ever more functionality is being added; user expectations regarding quality and reliability increase; an ever tighter integration between the physical processes being controlled and the embedded hardware and software is needed; and technological developments push towards networked, multi-processor and multi-core platforms. The added complexity materialises in the software and hardware embedded at the core of the systems. Important decisions need to be made early in the development trajectory: Which functionality should be realised in software and which in hardware? What is the number and type of processors to be integrated? How should storage (both working memory and disk storage) and transfer of data be organised? Is dedicated hardware development beneficial? How to distribute functionality? How to parallelise software? How can we meet timing, reliability, and robustness requirements? The decisions should take into account the application requirements, cost and time-to-market constraints, as well as aspects like the need to reuse earlier designs or to integrate third-party components.

Industries often adopt some form of model-based design for the software and hardware embedded in their systems. Figure 7.1 illustrates a typical process. Whiteboard and spreadsheet analysis play an important role in early decision making about design alternatives. System decompositions are explored behind a whiteboard. Spreadsheets are then used to capture application workloads and platform characteristics, targeting analysis of average- or worst-case utilisation of platform resources. They provide a quick and easy method to quantitatively explore alternatives from

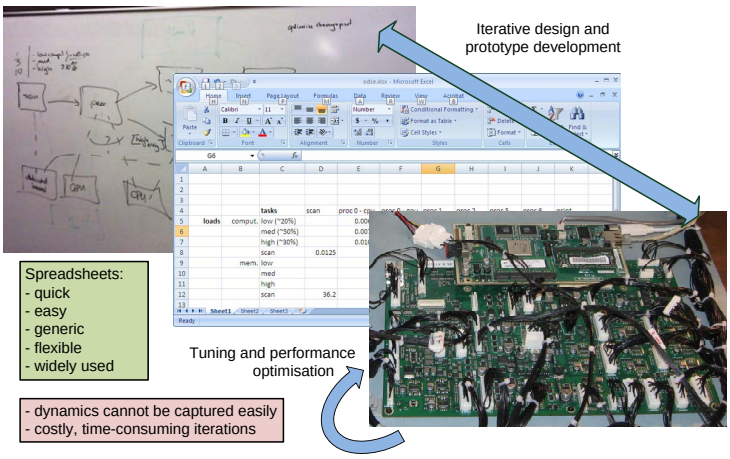

Fig. 7.1 Typical industrial design practice for embedded hardware and software: iterative design and development, intensively using spreadsheets, tuning functionality, and optimising performance in prototypes.

performance and cost perspectives, at a high abstraction level. Promising alternatives are then realised (using various design and coding tools), to validate and fine-tune functionality and performance at the level of an implementation model. Implementation models typically realise important parts of the functionality, they integrate real code, and may run on prototype hardware. The entire process may be iterated several times before arriving at the final result.

Design iterations through prototypes are time-consuming and costly. Only a few design alternatives can be explored in detail. The number of design alternatives is however extremely large. The challenge is therefore to effectively handle these many possibilities, without loosing interesting options, and avoiding design iterations and extensive tuning and re-engineering at the implementation level. Spreadsheet analysis is suitable for a coarse pruning of options. However, it is not well suited to capture system dynamics due to for example pipelined, parallel processing, data-dependent workload variations, scheduling and arbitration on shared resources, variations in data granularity, etc. (see Fig. 7.2).

Understanding and analysing the pipelined, parallel processing of dynamic streams of data is challenging. The relation between design parameters (such as the number and type of processing units, memory size and organisation, interconnect, scheduling and arbitration policies) and metrics of interest (timing, resource utilisation, energy usage, cost, etc.) is often difficult to establish. An important challenge in embedded-system design is therefore to find the right abstractions to support accurate and extensive design-space exploration (DSE).

This chapter presents an approach to model-driven DSE and a supporting tool set, the Octopus tool set. The approach targets an abstraction level that captures the important dynamics while omitting the detailed functional and operational behaviour. The abstractions bridge the gap between spreadsheet analysis and implementation

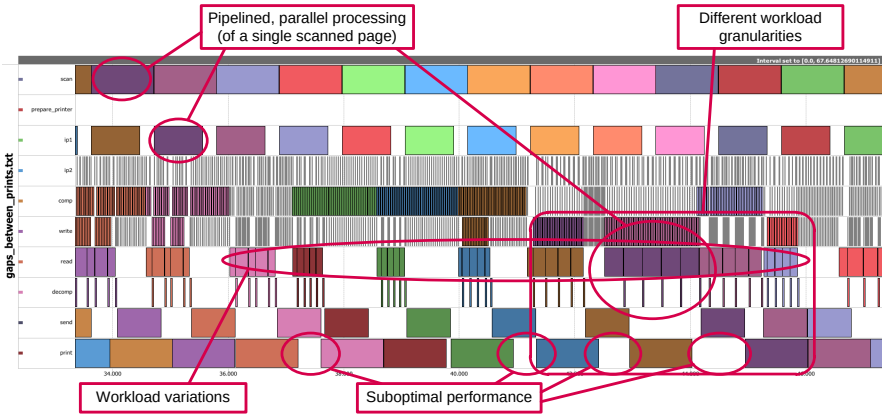

Fig. 7.2 A Gantt chart showing the execution of a print pipeline. Dynamics in the processing pipeline cause hick-ups in print performance due to under-dimensioning of the embedded execution platform. (Figure from [7])

models and prototypes. The approach is designed to specifically cope with the challenges of DSE.

An important characteristic of DSE is that many different questions may need to be answered, related to system architecture and dimensioning, resource cost and performance of various design alternatives, identification of performance bottlenecks, sensitivity to workload variations or spec changes, energy efficiency, etc. Different models may be required to address these questions. Models should be intuitive to develop for engineers, potentially from different disciplines (hardware, software, control), and they should be consistent with each other. Multiple tools may be needed to support the modelling and analysis.

Given these characteristics, our approach to address the challenges of model-driven DSE is based on two important principles: (1) separation of concerns and (2) reuse and integration of existing techniques and tools. The modelling follows the Y-chart paradigm of [6, 39] (see Fig. 7.3) that separates the concerns of modelling the application functionality, the embedded platform, and the mapping of application functionality onto the platform. This separation allows to explore variations in some of these aspects, for example the platform configuration or the resource arbitration, while fixing other aspects, such as the parallelised task structure of the application. It also facilitates reuse of aspect models over different designs. The tool set architecture separates the modelling of design alternatives, their analysis, the interpretation and diagnostics of analysis results, and the exploration of the space of alternatives (see Fig. 7.4). This separation is obtained by introducing an intermediate representation, the DSE Intermediate Representation (DSEIR), and automatic model transformations to and from this representation. This setup allows the use of a flexible combination of models and tools. It supports domain-specific modelling in combination with generic analysis tools. Multiple analyses can be applied on the same model, guaranteeing model consistency among these analyses; different analy-

sis types and analyses based on multiple models can be integrated in a single search of the design space. Results can be interpreted in a unified diagnostics framework.

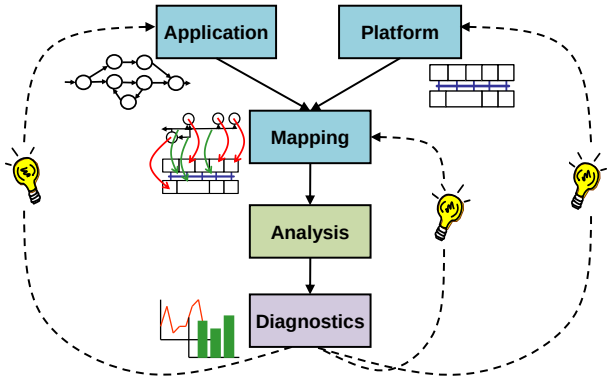

Fig. 7.3 The Y-chart paradigm for design-space exploration separates the modelling of application functionality, platform functionality, and mapping of application functionality onto the platform; after analysis and diagnostics, any of these aspects may be changed to explore alternatives (either automatically or interactively by a designer). (Y-chart: [6, 39]; figure from [8])

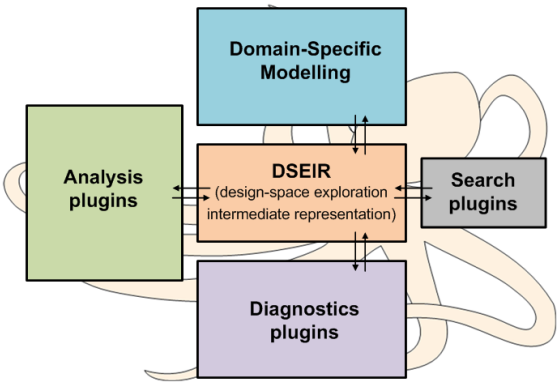

Fig. 7.4 Top view of the Octopus tool set. (Figure from [7])

Chapter overview. Section 7.2 provides an overview of the challenges we have encountered in modelling and analysis support for taking design decisions during early development. The experience draws upon our work in the professional printing domain, but the challenges are valid for a wide range of high-tech embedded systems. Section 7.3 explains the model-driven DSE approach we propose to handle these challenges. This section also surveys related work. To illustrate the possibilities for domain-specific modelling, Section 7.4 presents DPML, the Data Path Modelling Language, which is a domain-specific modelling language for the print-

ing domain. A DSE case study from the professional printing domain is introduced to illustrate DPML. Section 7.5 introduces the intermediate representation DSEIR, which is at the core of the Octopus tool set. Section 7.6 presents model transformations to a number of analysis tools. Section 7.7 illustrates the support integrated in Octopus for interpretation and diagnostics of analysis results. Section 7.8 briefly discusses some implementation choices underlying the tool set. Section 7.9 presents the results we obtained in several industrial case studies. Section 7.10 concludes.

Bibliographical notes. An extended abstract of this chapter appeared as [7]. The Octopus tool set was first described in [8]. Our philosophy behind model-driven DSE was originally presented in [69]. Sections 7.2 and 7.3 are based on [69]. Section 7.4 describing DPML is based on [65], which provides a more elaborate description of DPML. DSEIR, summarised in Sect. 7.5, is described in more detail in [69].

7.2 Challenges in Early Design

Making the right decisions early in the design process of a complex software-intensive embedded system is a difficult task. In this section, we discuss the challenges we faced while conducting several case studies at Océ-Technologies, involving the design of digital data paths in professional printers. These challenges are characteristic for other application domains as well.

Multi-functional printing systems perform a variety of image processing functions on digital documents that support the standard scanning, copying, and printing use cases. The digital data path encompasses the complete trajectory of the image data from source (for example the scanner or the network) to target (the imaging unit or the network).

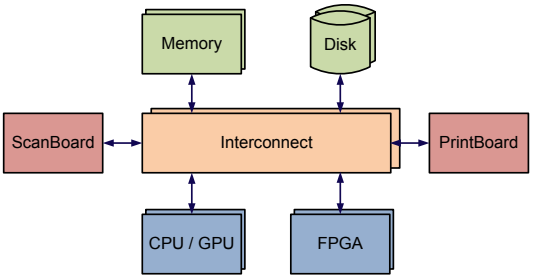

Fig. 7.5 A template for a typical printer data path architecture. (Figure from [69])

Data path platform template. Figure 7.5 shows a template of a typical embedded platform architecture for the digital data path of a professional printer. Several special-purpose boards are used to perform dedicated tasks, typically directly related to the actual scanning and printing. For computation, the data path platform

may provide both general-purpose processors (CPUs, GPUs) and special-purpose FPGA-based boards. RAM memory and hard disks are used for temporary and persistent storage. The components are connected by interconnect infrastructure (e.g. PCI, USB). The architecture template shows the components needed for the digital image processing, leaving out user controls, network interfaces, etc. Note that the template is in fact generic for almost any modern software-intensive embedded system.

Printer use cases. Each printer needs to support dozens of use cases. The standard ones are scanning, printing, copying, scan-to-email, and print-from-disk. Each use case typically involves several image processing steps such as rendering, zooming, rotating, compressing, half-toning, etc.; these steps may need several components in the platform, and different choices for implementing these steps may be possible. Moreover, print and scan use cases can be mixed. They can also be instantiated for documents with different paper sizes, numbers and types of pages, etc. It is clear that this results in an explosion of possibilities.

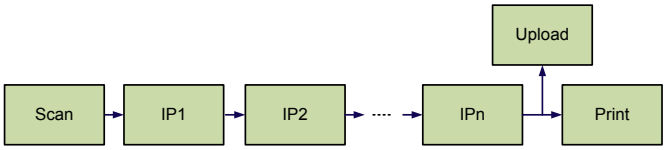

Fig. 7.6 An example printer use case: copying.

As an illustration, we sketch the copying use case in some more detail; see Fig. 7.6. After scanning, each page is first processed by the ScanBoard (that implements the Scan task) and then further processed in several image processing steps (IP1…IPn); the resulting image is then printed by the PrintBoard (executing the Print task). The image processing steps need to be done on some computation resource. Intermediate results are stored in memory and/or on disk. The latter is done for example to allow more than one copy of the document to be printed or to cope with errors. Uploading a processed image to disk can be done in parallel with any further processing.

Questions in data path design. The ScanBoard and PrintBoard are typically the first components to be selected for a specific printer. They determine the maximum possible scan and print speeds in pages per minute. The rest of the data path should be designed in such a way that these scan and print speeds are realised at minimum cost.

Typically, a number of questions need to be answered early in the development trajectory. Which types of processing units should be used? How many? What clock speeds are needed? What amount of memory is needed? Which, and how many, buses are required? How should image processing steps be mapped onto the resources? Other questions relate to scheduling, resource allocation, and arbitration. What should be the scheduling and arbitration policies on shared resources? What

are the appropriate task priorities? Can we apply page caching to improve the performance? How to allocate memory in RAM? How to share memory? How to minimise buffering between tasks? How to mask workload variations? Is the memory allocation policy free of deadlocks?

The data path design should take into account all basic use cases, as well as combinations such as simultaneous printing and scanning. It should also take into account different job types (text, images), paper sizes, etc. The design should be such that no bottlenecks are created for the most important use cases (normal printing, scanning, and copying, on the default paper size for normal jobs). Performance can be traded off for costs for use cases that occur less frequently though. DSE should also provide insight in these trade-offs. Furthermore, printing products typically evolve over time. This raises questions such as what is the impact of a new scan- or print board with higher specs on an existing data path design. It is clear that DSE complexity is large and that quantifying all the mentioned aspects early in the design process is challenging.

Modelling dynamics. Modelling the above use cases for DSE is possible with a spreadsheet at a high level of abstraction as a first-order approximation. Spreadsheet estimates, however, may lead to over- or under-dimensioning of the ultimate design, which is costly to repair in later design stages. The cause for over- or under-dimensioning is the difficulty to capture various dynamic aspects of software-intensive embedded systems in a spreadsheet. First, there are various sources of variability. The complexity of a page to be printed, the size of a compressed page, and the execution time on a general-purpose processor are all stochastic and rarely exactly predictable. Second, the flow of information is often iterative or conditional. An example of a conditional flow is a smart-storage heuristic that takes a page from disk only if it is not still in memory. Third, pipelined and parallel steps in a job and simultaneously active print and scan jobs may dynamically interact on shared resources such as memory, buses, and shared processors. Finally, scheduling and arbitration policies are often crucial for performance, but result in dynamic behaviour that is hard if not impossible to model in a spreadsheet-type model.

Mixed abstraction levels. Although many image processing steps work on pixels or lines, parts of most use cases can be accurately modelled at the page level. The throughput of the data path in images per minute is also the most important metric of interest. A mixture of abstraction levels may be needed to achieve the required accuracy while maintaining analysis efficiency. FPGAs for example come with limited memory sizes. Only a limited number of lines of a page fit in FPGA memory. The page level thus becomes too coarse and a finer granularity of modelling is needed. Modelling complete use cases at the line or pixel level would make most analyses intractable though; appropriate transitions between abstraction levels are needed.

Variety in analysis questions. The typical analysis questions in the list of DSE questions above may, in theory, all potentially be answered by a generic modelling and analysis tool; it is clear, however, that the various types of DSE questions may be of a very different nature. Deadlock and schedulability checks, for example, are best done using a model checker. Performance analysis would typically be done with

analytic models, like spreadsheets, for a coarse evaluation and simulation for a more refined analysis that takes into account the dynamics in the system. Low-level FPGA buffer optimisation can be done with fast, yet restrictive, dataflow analysis. This variety in analysis questions suggests the use of different tools. This does require the development of multiple models though, leading to extra modelling effort and a risk of model inconsistencies and interpretation difficulties. Ideally, one master model would form a basis for analyses performed with different techniques and tools.

Model parametrisation. There is a need to support a high degree of parametrisation of models: Multiple use cases need to be captured, each of them with many variations; models are needed for various DSE questions; design decisions may need to be reconsidered to cope with late design changes; and to speed up development, there is a clear wish to reuse models across variations of the same product, both to allow product customisation and to support the development of product families. The desired parametrisation goes beyond simple parameters capturing for example workloads, task execution times, and memory requirements; they should also cover for example the flow of use case processing, communication mechanisms, platform resources, and scheduling and arbitration policies.

Customisation for the printer domain. The basic principles of printer platforms and printer use cases are not rapidly changing. Having the models written in a printer-specific language, and maintaining a library of those models, would drastically decrease the modelling effort for new printers, reduce modelling errors, and improve communication and documentation of design choices. The design of a domain-specific language for the printer domain is challenging. On the one hand, we want a simple language, which only contains constructs that are needed to describe the current designs. On the other hand, it should also be possible to use (simple extensions of) the language to describe the designs of tomorrow.

7.3 Model-Driven Design-Space Exploration

The previous section clarified the challenges in early DSE. In this section, we first identify the main benefits of a model-driven approach to tackling these challenges. We then set out the key ingredients of our approach to model-driven DSE. Along the way, we survey methods, languages, and tools that fit in such an approach. The following sections then elaborate on the Octopus tool set that is being developed to support the model-driven DSE approach and that integrates several of the surveyed methods, languages, and tools.

The benefits of model-driven DSE. The ultimate objective of model-driven DSE is to reduce development time (and thereby time-to-market), while maintaining or improving product quality. This is achieved by appropriate modelling and analysis during early development. Models should capture the essential system dynamics without including unnecessary details. Only then, effective exploration of

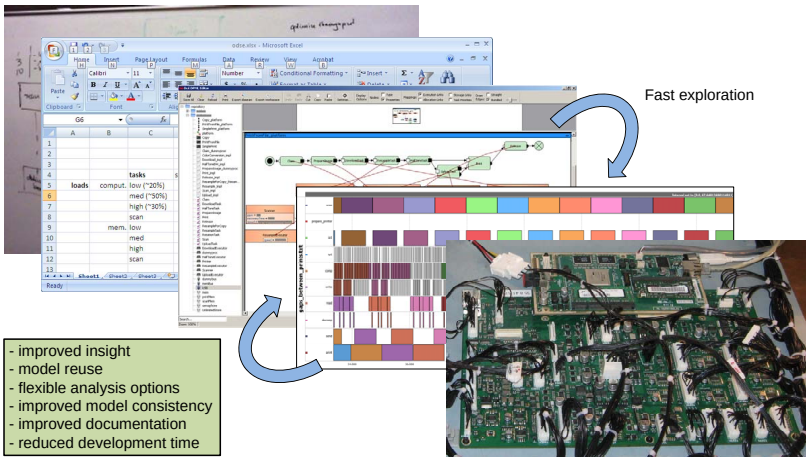

Fig. 7.7 Model-driven design-space exploration.

design alternatives is feasible. Figure 7.7 visualises the model-driven DSE approach and summarises the targeted benefits. With appropriate modelling and tool support, (1) insight in system dynamics and design alternatives improves, (2) models are re-usable within the product development trajectory and across developments, (3) it becomes feasible to apply different analyses, (4) different models may be used while safeguarding their consistency, and (5) documentation improves by using the models themselves as the design documentation. In combination, these benefits lead to (6) the intended reduction in development time.

Separation of concerns. To address the challenges outlined in the previous section and to realise the above-mentioned benefits, we propose a rigourous separation of concerns.

First, the tool set organisation should separate the modelling, analysis, diagnostics, and search activities, as already illustrated in Fig. 7.4. Modules for each of these activities are decoupled through an intermediate representation, DSEIR (DSE Intermediate Representation; see Sect. 7.5). Such an organisation realises the required flexibility in modelling and analysis needs. The use of an intermediate representation allows reuse of analysis techniques and tools across different models and in combination with different modelling environments. Model consistency is ensured by appropriate model transformations to and from the intermediate representation. The challenge is to develop an intermediate representation that is sufficiently rich to support DSE but not so complex that it prohibits model transformations to various analysis techniques and tools. These model transformations should preserve precisely defined properties, so that analysis results from different tools can be combined and results can be interpreted in the original model.

Second, modelling should follow the Y-chart philosophy [6, 39]. This 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 (as

already illustrated in Fig. 7.3). Diagnostic information is then used to, automatically or manually, improve application, platform, and/or mapping. This separation of application, platform, and mapping is important to allow independent evaluation of various alternatives of one of these system aspects while keeping the others fixed the others. Often, for example, various platform and mapping options are investigated for a fixed set of applications. DSEIR separates the application, platform, and mapping modelling. In combination with the tool set organisation illustrated in Fig. 7.4, DSEIR thus supports the Y-chart philosophy.

Application-centric domain-specific modelling. A key challenge in modelling for DSE is to fully take into account the relevant system dynamics such as realisable concurrency, variations in application behaviour, and resource behaviour and sharing. Given the complexity of the DSE process, a modelling abstraction level is needed that abstracts from implementation details but is more refined than typical spreadsheet-type analysis. Modelling should be simple and allow efficient and accurate analysis.

Another important aspect in modelling for DSE is that the abstractions need to appeal to the designer and adhere to his or her intuition. Domain-specific abstractions and customisation should therefore be supported. Modelling should furthermore be application-centric. The application functionality and the quality (performance, energy efficiency, reliability) with which it is provided is what is visible to users and customers. Application functionality should therefore be leading, and the models should capture all behaviour variation and all concurrency explicitly. We propose to model platforms as sets of resources that have no behaviour of their own; their purpose is only to (further) restrict application behaviour and to introduce proper timing. This leads to simple, predictable, and tractable models. Scheduling and mapping of applications onto resources can then be unified into the concept of prioritised dynamic binding. If needed, complex resource behaviour (work division, run-time reconfiguration, etc.) can be modelled through (automatic) translations into application behaviour.

A variety of modelling environments and approaches in use today, either in industry or in academia, can support the envisioned modelling style. We mention some of them, without claiming to be complete: AADL [1], DPML [65], Modelica [47], Ptides [19], Ptolemy [21], MATLAB/Simulink [45], SysML [64], SystemC [51], UML [72], and UML-MARTE [73]. The mentioned environments often target different application domains and/or different system aspects. Figure 7.8 positions these modelling approaches in the architectural framework of the Octopus tool set. DPML, the Data Path Modelling Language, is discussed in more detail in Sect. 7.4.

Analysis, search, diagnostics. The previous section illustrated the variety of design questions and challenges that are typically encountered early during development. No single tool or analysis method is fit to address all these questions. We foresee the combined use of different analysis tools in one DSE process. A wide variety of, mostly academic, but also some commercial, tools is available that can be used to support DSE.

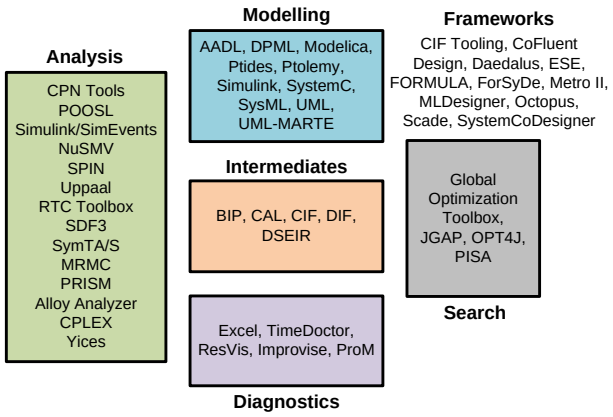

Fig. 7.8 Methods, languages, and tools that fit in the top-level architectural view of Fig. 7.4. Every entry is only mentioned where it fits best in the architectural view (even though it may fit in other places as well). (Figure adapted from [69])

For quick exploration and performance optimisation, discrete-event simulators such as CPN Tools [34], POOSL [66], and Simulink/SimEvents [45, 44] are suitable. Model checkers such as NuSMV [50], SPIN [25], and Uppaal [10] can be used for functional verification, protocol checking, and schedule and timing optimisation. Model checkers may not be able to cope with the full complexity of modern embedded systems, but they may play an important role in verifying and optimising critical parts of the system. Yet other tools, such as the RTC Toolbox [57], SDF3 [62], and SymTA/S [63], are suited for timing analysis of data-intensive system parts, such as image processing chains.

Questions regarding performance, reliability, and schedulability under soft deadlines can be answered by increasingly popular probabilistic model checking techniques, using tools like PRISM [54] and MRMC [37]. These techniques enhance the expressivity of regular model checking, allowing for more realistic modelling of aspects such as arrival rates and failure times. Scalability of these techniques to realistic analysis problems remains a challenge though.

In recent years, also constraint programming and SAT/SMT solvers have gained popularity. With the rise of more powerful computers and improvements in the techniques themselves, tools like CPLEX [28], Alloy Analyzer [32], and Yices [80] are increasingly often used to find optimal or feasible solutions for system aspects such as resource bindings or schedules.

The Octopus tool set has links to three analysis tools, namely CPN Tools, Uppaal, and SDF3. The model transformations that realise these links and the intended use of these tools in the Octopus context are discussed in Sect. 7.6.

Besides support for evaluation of metrics for design alternatives or for the optimisation of parts of the system, we also need support to explore the large space of design alternatives. The MathWorks Global Optimization Toolbox [43] supports a wide variety of customisable search algorithms. The JGAP library [36] is a Java

library for developing genetic search algorithms. OPT4J [42] and PISA [11] are customisable genetic search frameworks that support DSE.

Most of the tools mentioned above already give good diagnostic reports, which in many cases can be successfully converted and interpreted in the original domain. Microsoft Excel is also a useful tool in this context. Visualisation of Gantt charts (using tools such as TimeDoctor [68] or ResVis [59], from which the screenshot of Fig. 7.2 is taken) helps understanding the dynamic behaviour of design alternatives. Sophisticated mining and visualisation is possible with Improvise [31] or ProM [55]. Section 7.7 presents the diagnostic support as it is developed in the Octopus context, which includes Gantt chart visualisation through ResVis.

Intermediate representation: flexibility, consistency, customisation. It cannot be expected that designers master the wide variety of modelling languages and tools mentioned so far, and apply them in combination in DSE. To successfully deal with integration, customisation, and adaptation of models, as well as to facilitate the reuse of models across tools and to ensure consistency between models, we foresee the need for an intermediate representation to connect different languages and tools in a DSE process. Such an intermediate representation must in the first place be able to model the three main ingredients of the Y-chart (application, platform, mapping) in an explicit form. It should not have too many specific constructs to facilitate translation from different domain-specific modelling languages and to different target analysis tools, yet it must be powerful and expressive enough to accommodate developers. A good balance between modelling expressiveness and language complexity is needed. Besides the Y-chart parts, the intermediate representation must provide generic means to specify sets of design alternatives, quantitative and qualitative properties, experimental setups, diagnostic information, etc., i.e. all ingredients of a DSE process. The intermediate representation should have a formal semantic basis, to avoid interpretation problems and ambiguity between different models and analysis results. The intermediate representation does not necessarily need execution support, because execution can be done through back-end analysis tools. Intermediate representations and languages like BIP [9], CAL [20], CIF [74], DIF [27] and the intermediate representation DSEIR (see Sect. 7.5) underlying the Octopus tool set are examples of languages that can be adapted to fully support model-driven DSE as sketched in this section.

A DSE tool set. To realise the goals set out, it is important to provide a flexible tool set implementation. We propose a service-based implementation of the tool set architecture of Fig. 7.4. Modules should communicate with other modules through clean service interfaces. Domain-specific modelling tools with import/export facilities to DSEIR are in the modelling module. The analysis module provides analysis services such as performance evaluation, formal verification, mapping optimisation, schedulability analysis, etc. The diagnostics module provides ways to visualise analysis results and gives high-level interpretations of system dynamics in a way intuitive to system designers. The search module contains support for search techniques to be used during DSE. Information flows between the modules go through the DSEIR kernel module that implements the intermediate representation.

There are several frameworks and tool sets that support DSE, or aspects of it, for various application domains. Examples are CoFluent Design [15], Daedalus [49], ESE [75], FORMULA [33], ForSyDe [58], METRO II [17], MLDesigner [46], SCADE [22], and SystemCoDesigner [38]. The large number of available languages, tools, and frameworks are a clear indication of the potential of high-level modelling, analysis, and DSE. The Octopus tool set, described in more detail in the remainder of this chapter, is closest to the views outlined in this section. Octopus explicitly aims to leverage the combined strengths of existing tools and methods in DSE. Its service-based implementation is discussed in Sect. 7.8.

7.4 DPML: Data Path Modelling Language

The entry point of our tool chain is the modelling module (see Fig. 7.4). Models can be developed in a domain-specific language (DSL), that functions as a front end for the tool chain. For modelling printer data paths, we have designed DPML (Data Path Modelling Language). This section first introduces a typical DSE case study as a running example. It then presents the design goals for DPML, followed by an overview of DPML and a presentation of the main DPML concepts along the lines of the Y-chart separation of concerns (see Fig. 7.3).

7.4.1 Running Example: High-End Colour Copier

The scan path of a copier is the part of the data path that receives data from the scanner hardware, processes them, and stores them for later use (e.g. sending them to the printer hardware or to an e-mail account). Figure 7.9 shows the high-level architecture of a part of the scan path in a high-end colour copier. Its structure follows the Y-chart approach. The application consists of six tasks: a step that downloads image data from the scanner hardware, four image processing steps IP1, ..., IP4, and a step that writes the processed image data to disk. The tasks pass image data through various statically allocated buffers (i.e. the buffer slot size is constant and both the size and the number of slots are determined at design time). Note that task IP4 reads from and writes to the same buffer slot. The platform consists of a general-purpose board with CPUs/GPUs and potentially several dedicated boards, for example for the interface with the print engine (not shown in the figure). The mapping of the tasks and the buffers to the platform is depicted by the colour-star combination. Steps IP1, IP2, and IP4 all use the CPU, whereas the other steps each have their own computational resource.

There are several important aspects that complicate the modelling and analysis:

- Various steps use different data granularities (indicated in the figure by the different widths of the arrows to and from the buffers). The download step writes the data of a complete image in a buffer slot of buffer 1. Step IP1 processes these

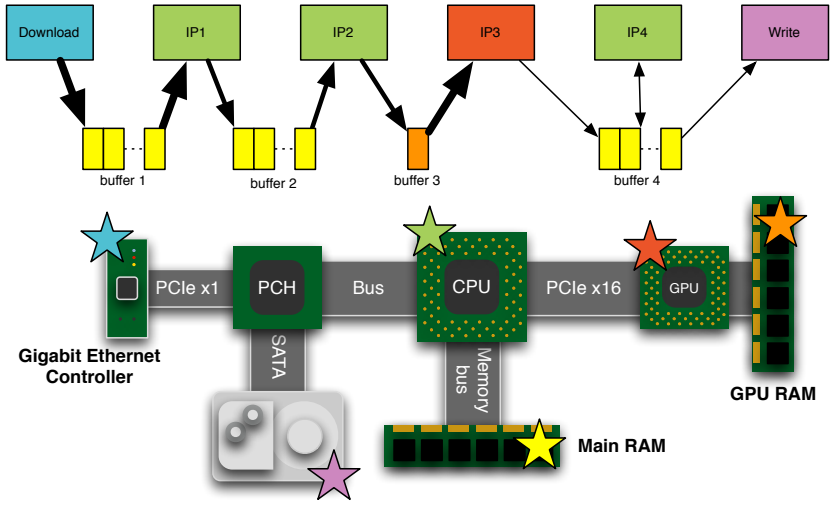

Fig. 7.9 High-level architecture of a part of the scan path in a high-end colour copier.

data and outputs the result in ten so-called *bands*. Step IP2 processes each band and appends it to the single slot of buffer 3. As soon as the data of a complete image are present in buffer 3, step IP3 processes them and writes the result in finer grained bands to buffer 4. Steps IP4 and *Write to disk* process these bands individually.

- Buffers 1, 2, and 4 can have multiple slots which allows pipelining of the processing steps of consecutive images.
- The scheduling on the CPU is priority-based preemptive.
- The execution times of the steps are not known exactly, and often heavily depend on the input image and on the result of the algorithms in the steps that change the data size (e.g. compression).

The main performance indicator of the data path is *throughput*. It is important that the data path is not the limiting factor of the machine. For cost reasons, the scan hardware should be fully utilised. Another, albeit less important, performance indicator is the total amount of main memory that buffers 1, 2, and 4 consume. Since memory is a scarce resource, the buffers should not be over-dimensioned. The DSE question is therefore as follows:

Minimise the amount of memory allocated to buffers 1, 2, and 4 while retaining a minimum given throughput.

7.4.2 The DPML Design Goals

DPML is intended to support modelling for DSE of printer data paths. When designing DPML, four goals were kept in mind:

- First, DPML must be particularly suited to analyse the *speed* (or throughput) of data path designs. This means that all information necessary for obtaining the speed of a data path must be present, but behavioural issues that do not influence speed may be abstracted away. The data sizes of images and image parts being manipulated and transferred play a dominant role.
- Furthermore, DPML has to be *expressive* and *flexible*. This means that it must be possible to express a wide variety of models, with different behavioural and structural properties. This is important, because we cannot always foresee what kinds of designs engineers may want to analyse in the future, or what other purposes (than analysing speed) may be found for DPML models. Therefore, it must be easy to model many things in DPML, and it must also be easy to change DPML to add more features. This requirement is essential if the tool chain is to be a sustainable solution.
- DPML has to *closely match the problem domain*. This means that all elements commonly found in printer data paths must be well supported and easy to model. The most visible example of this is the concept of pages flowing through the steps of the application. Without this, even simple designs would require considerable modelling effort and thus a steeper learning curve for people using the tools for the first time. This may in turn impact the adoption of the tool set as a means to improve the data path design process.
  Important are also the features that DPML does *not* have; features that would make it a more generic specification language, such as support for caches, the ability to fully specify the behaviour of steps, the possibility to specify real-time deadlines, or the ability to fine-tune a task scheduler. Leaving out such features makes DPML a simple language, in which models of commonly occurring data path designs are not significantly more complex than what an engineer would draw in an informal sketch of the same design.
- Finally, DPML has to support *modular* designs. This way, parts, or elements that are used in multiple designs can be reused, thus saving modelling time. Additionally, a modular setup allows engineers to share knowledge obtained during design or engineering (such as the actual speed of a hardware component, or the rationale behind a design decision) with engineers in other projects who may use some of the same parts.

7.4.3 DPML Overview

DPML is a combined visual and textual language. The visual parts outline the coarse structure of a data path design, such as the series of processing steps that a data path

may be required to perform, and the component layout of a hardware platform. These are represented visually so that, even in large models, it is easy to get a good overview. The textual parts are used to express all details of every element in a data path design, such as the behaviour of a step or the capacity of a memory. These details are expressed with a custom, text-based language so that it is easy to add new constructs and features.

Structurally, a complete DPML model consists of the three distinct components of the Y-chart paradigm:

- An **application**, which functionally describes a series of steps that a data path may be required to perform.
- A **platform**, which describes the various hardware components that a data path consists of and how they are laid out.
- A **mapping** between these two, which describes which hardware components are used by which steps, and how.

DPML models can be edited in a custom editor that is based on the open source Qt library [56]. A single DPML model is contained in multiple small files, each of which describes a reusable element. A textual DPML element is stored as a plain text file and visual DPML elements are stored in a custom XML format. To facilitate the analysis and further conversion of DPML models, e.g. to the Octopus intermediate representation DSEIR, these files are converted to a simpler data format, DPML Compact. DPML Compact models can be simulated using a native simulator. The advantage of having this compact format is that changes in and additions to the language do not affect the simulator and the model transformations as long as DPML Compact remains unchanged.

7.4.4 DPML: The Application View

Figure 7.10 displays the application component of our running example described in DPML. The model captures the pipelined processing of scan jobs consisting of any number of scanned pages. Every rounded rectangle in Fig. 7.10 is a **step**, which is a single image processing operation that the data path must perform on a single image or part of an image. Each step has two or more **pins**, the squares or circles on the sides.

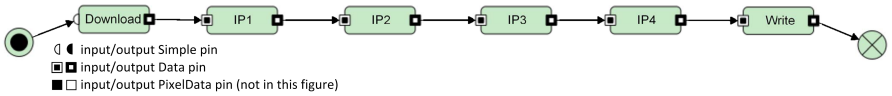

Fig. 7.10 A DPML application.

Pins can have three possible types: `Simple`, `Data`, and `PixelData`. Visually, a `Simple` pin is a semicircle, a `Data` pin is a square, and a `PixelData` pin is a

square with a small square in the middle. Every `PixelData` pin also is a `Data` pin, and every `Data` pin is also a `Simple` pin. When a step has an output `Data` pin, this means that this step *produces data* at that pin. If it is a `PixelData` pin, this additionally implies that the data produced represent a bitmap image. Similarly, when a step has an input `Data` pin, it means that this *consumes data* at that pin.

Arcs between steps indicate data dependencies. They further determine the execution order of the application. By default, a step can only start when for all its input pins, the preceding step has completed. This explains the need for `Simple` input pins; no data are consumed or produced on those pins, but they can be used to fix the execution order between steps.

Because a printer data path is all about image processing, we assume that we can model the duration of a single image processing step as a function of the size of the image data, the speed and availability of all resources used, and predefined information about the pages that are to be processed. Most notably, we assume that it does not depend on the *precise content* of the image. This assumption is important, because it means that instead of formally specifying all aspects of a step behaviour, it is sufficient to just specify the *data sizes* that it produces.

In DPML, a **task** is used to describe a single operation that we may want a data path to perform. Each step in an application is in fact an *instantiation* of such a task: A step behaves exactly as dictated by the task, and each step corresponds to exactly one task. It is, however, possible for multiple steps to belong to the same task. Multiple compression step instances of the same task may for example occur in a single image processing pipeline. Because a step cannot exist without a task, the set of available tasks defines the set of operations we can use in an application. The relationship between tasks and steps is therefore somewhat comparable to the relationship between classes and objects in object-oriented programming languages.

Tasks are stored as small text files with content as shown in Fig. 7.11. A task definition has three parts: a header, a set of pin declarations, and a set of properties. The header defines the name of the task, in this case “Resample”. The pin declarations define the number of input and output pins, their names, and their data types. The Resample task takes a raster image and produces a raster image, so there is one input pin and one output pin, both of type `PixelData`. The properties constitute the functional description of the behaviour of a task. Because we assume that the actual content of the image produced does not matter, we only need to determine the size of the output image (so in fact we are only describing a very small part of the required behaviour, focusing on resource usage and performance aspects).

In the example of the Resample task, the width and length of the output image depend on the width and length of the input image as well as a user setting, the zoom factor. Because it is possible for a single scan job to require some pages to be zoomed and some pages not, the zoom setting is looked up as a property of the current page.

Tasks can specify more properties than just the sizes of its output images, such as a boolean condition that must be true for a task to be able to start. All of these other properties are optional.

```
task Resample
{
  inpin  in:  PixelData;
  outpin out: PixelData;

  out.width    = in.width  * page.zoomFactorX;
  out.length   = in.length * page.zoomFactorY;
  out.bitdepth = in.bitdepth;
  precondition = true; //optional and redundant
}
```

Fig. 7.11 An example of a task in DPML. (Figure from [65])

7.4.5 DPML: The Platform View

A platform defines the hardware necessary to perform the steps in a data path. Such hardware typically includes processors, memories, hard drives, caches, cables and buses, and of course the actual printing and scanning components. In DPML, these components belong to a set of three resource types:

- A **memory** is something that can store and retrieve data, so it includes hardware such as RAM chips and hard drives. A memory has a particular capacity, which is its only limiting factor.
- A **bus** is something that can transfer data. A bus typically has a maximum bandwidth (the maximum number of bytes it can transfer per second).
- An **executor** is something that can execute a step and has some processing speed. Executors are subdivided into processors, scanners, and printers for clarity, but from a semantics point of view there is no difference between a processor, a scanner, and a printer in DPML.

With just these blocks, we can create sufficiently realistic platform models for analysing the performance of a data path design. The platform of our running example looks like in Fig. 7.12.

Memory blocks are shown as rounded rectangles, bus and executor blocks as rectangles. Buses are the only components that can limit data transfer speed. Thus, the `Disk` memory, which models a hard drive, can in fact read and write data infinitely fast. The `SATA` bus models both the real SATA bus by means of which the hard drive is connected to the PC motherboard and the hard drive's inherent maximum read/write speed. Note that the model represents the main RAM and GPU RAM memories in the form of the (statically allocated) buffers `Main_RAM_1`, `Main_RAM_2`, `Main_RAM_3`, and `GPU_RAM`; the latter are the actual resources that tasks need to compete for.

A line between blocks in a platform model is called a **connection**, meaning that the two (or more) blocks are *directly* connected to one another. Connections limit the possible routes by which data can flow through the platform. An additional requirement is that, on a route between an executor block and a memory block, there may only be (one or multiple) bus blocks.

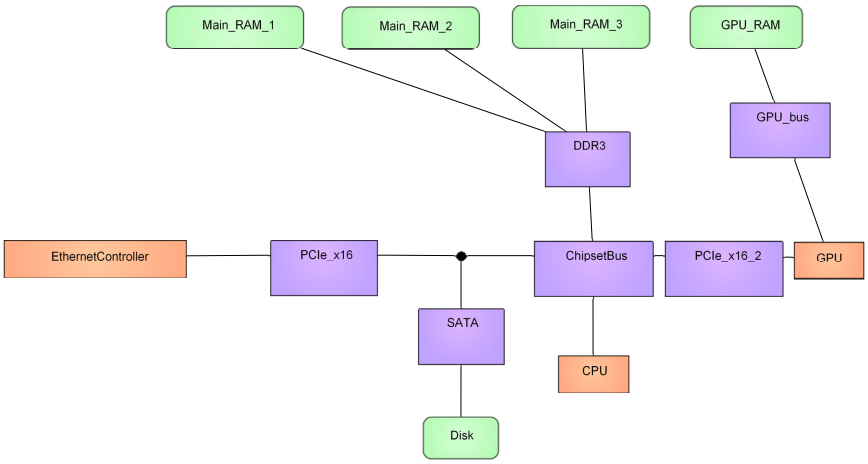

Fig. 7.12 A platform model in DPML.

Resource blocks have properties, much in the same way as the steps of an application have behaviour. For example, properties of resource blocks include the bandwidth of a bus and the capacity of a memory. Analogously to steps and tasks or to objects and classes, the properties of resource blocks are specified in small chunks of code called **resources**. A resource describes that a particular piece of hardware exists and has some particular properties. For example, a resource may describe a particular Intel processor. A resource block based on that resource, describes that such an Intel processor is used in an actual hardware platform. A resource block cannot exist without an associated resource, and there can be multiple resource blocks based on a single resource. Resources are typically simpler in structure than tasks. Many resources only have one or two properties.

```
processor CPU
{
  multitask = true;
  speed = 1G; //1 billion operations per second
}

bus GPU_Bus
{
  transferSpeed = 5M; //5 million bytes per second
}

Memory Main_RAM_1
{
  capacity = 2G; //2 billion bytes
}
```

Fig. 7.13 Some example resources.

As shown in Fig. 7.13, each resource type has its own (small) set of properties. The `transferSpeed` property for buses and the `capacity` property for memories always have the same unit: they are expressed in bytes per second and in bytes, respectively.

DPML platform models only have buses, memories, and executors. This means there are no more specialised versions of such resources, such as caches or hard drives. It turns out that, currently, such elements are not needed.

Recall that a data path is a component that performs image processing and transfer operations. This allows us to make the following assumptions:

1. Data transferred between steps are *new*, i.e. a step has not recently read or written exactly the same image.
2. Data are read and written linearly, i.e. in a single stream of ordered bytes.
3. The amount of working memory needed for processing an image is small.

Assumptions 1 and 2 imply that caches do not influence processing speed when an image is read from memory, because every chunk of image data read constitutes a cache miss. Additionally, because of Assumption 3, we can assume that reads and writes to the internal memory used by the image processing steps (such as local variables) always constitute a cache *hit*, i.e. that they seldom go all the way to the actual memory block.

Because data are written and read linearly (Assumption 2), we can ignore the fact that physical hard drives are relatively slow when reading randomly offset data. Compared to the time spent reading relatively large chunks of sequentially stored bytes (image data), the *seek time* needed to move the read head to the right position is negligible.

Note that it is not impossible to design a data path in which one or more of the above assumptions do not hold. The analysis of such a DPML model may yield significantly different results than the real data path would. Therefore, it is important that DPML users are aware of these assumptions. If it turns out that some assumptions are invalid more often than not, additional features may be added to DPML to overcome this issue. Note that due to the modular and extensible structure of DPML, it is relatively easy to do so if the need arises. New properties for memory resources that describe, for instance, a hard drive's random seek penalty and its fragmentation state may be used to estimate seek times penalties if deemed significant. Similarly, there is no structural reason why a fourth resource type, such as a cache, could not be added to the language if necessary.

7.4.6 DPML: The Mapping View

A mapping defines how an application relates to a platform. In a mapping, we specify which steps run on which executor blocks, which data are stored where, and which memory claims and releases are performed. Like applications and platforms, mappings are partly textual and partly graphical.

DPML mappings have three different kinds of **links** by which elements are mapped onto one another: storage links, allocation links, and execution links. Visually, a mapping is simply displayed by an application and a platform shown alongside one another, and links are represented as arrows between the application and the platform.

Storage links specify where the output data on each `Data` output pin of a step have to be stored for a subsequent step to be able to read and process them. A storage link is thus a link between output `Data` pins and memory blocks. Figure 7.14 shows how we can model this in DPML for our running example.

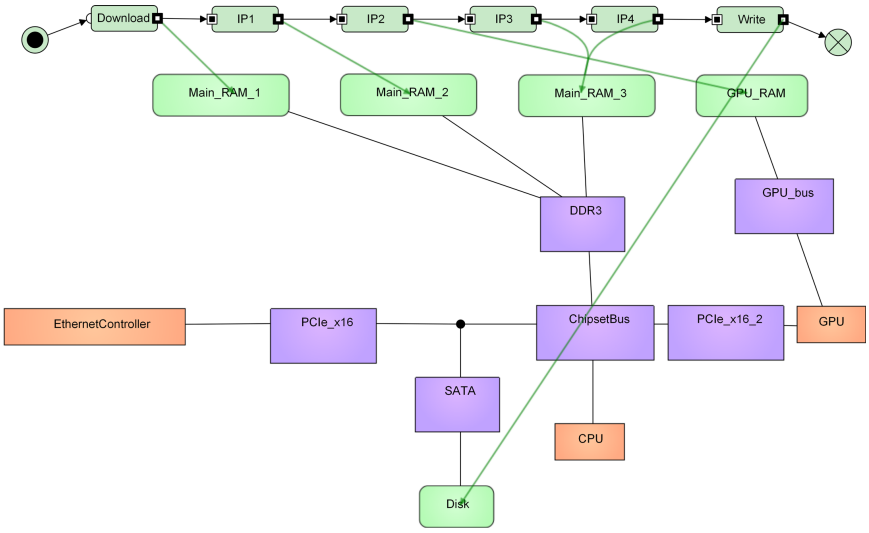

Fig. 7.14 Storage links in the mapping assign memory blocks to `Data` output pins.

Allocation links are used to keep track of memory claims and releases. Before data can be written to memory, it must first be allocated. This is important, because if a step cannot start because memory is full, the step must be blocked until sufficient memory is available for its output `Data` pins.

Fig. 7.15 Some allocation and release links and their short-hand notation.

There are two kinds of allocation links, see Fig. 7.15: claim links (blue) and release links (green). They are responsible for claiming and releasing the memory block, respectively. Even though links are part of the mapping, they are drawn entirely in the application; this is because instead of saying “step A claims memory

B", we say "step A claims the memory needed for output pin C". Using the storage links, analysis tools can then determine which memory block that pin is mapped to, and using the properties of the task associated to the output pin's step, it can be determined how much memory should be claimed.

The allocation links may create a busy picture that is difficult to oversee. Therefore, DPML provides two shorthand rules: if a `Data` output pin has no claim link, then the step that the pin belongs to is assumed to perform the claim. Similarly, if it has no release link, then the following step is assumed to perform the release. The second rule can only be applied if there is only a single step that consumes the data produced by the `Data` output pin. If there are more than one (or zero) following steps, a release link should always be drawn.

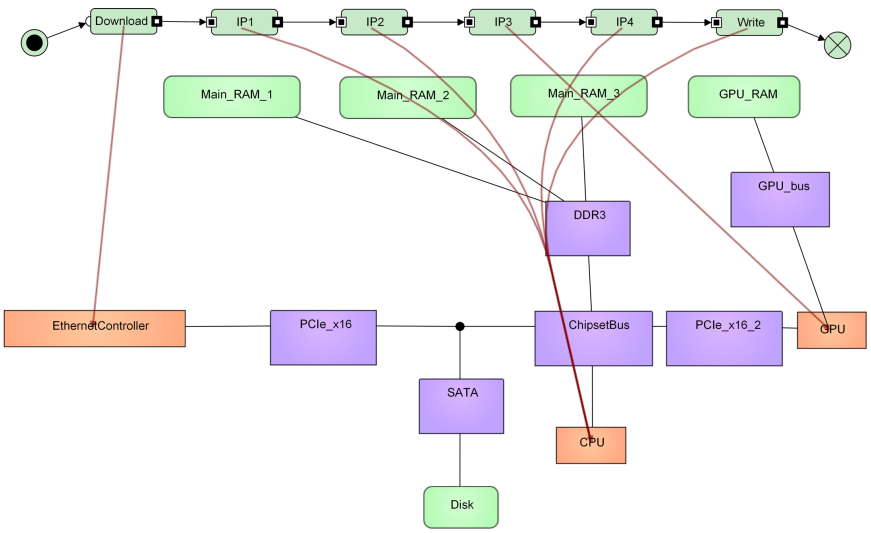

Fig. 7.16 Execution links.

Execution links, finally, describe which steps run on which executor blocks. Like storage links, they are drawn as simple arrows from steps to executor blocks. Every step can have at most one execution link, but an executor can be associated to any number of execution links.

Note that it is allowed for a step to not have an execution link, but only if the step has no `Data` pins. If a step consumes or produces data, then this means that data are being transferred from or to a memory, via some buses, to or from an executor block. Only by means of an execution link, this route can be computed.

Unlike storage links, each execution link has two additional properties: an associated *implementation* and a *priority*. Each (red) arrow between steps and executor blocks in Fig. 7.16 is an execution link.

An **implementation** describes *how* a step can be mapped onto an executor block. Because we are interested in the speed of the data path, an important property of a

single step is its duration. A step's speed is typically limited either by the available processing power, or by the available bus capacity for reading and writing its input and output data. In order to know which of the two limits a step's speed, we need to compute both. A step's bus usage can be derived from the size in bytes of its inputs and outputs and from the platform layout. Computing a step's processing speed, however, requires some more information from the user.

The duration of a single step depends on properties of the processor, on properties of the data (such as the image width and length) and on the particular implementation of the step. A DPML implementation captures this dependency in the property `processingDuration`. This property specifies the amount of time that a step is expected to take to process a single block of image data, given the current circumstances. As this time usually depends on some property of the input and/or output data as well as the amount of processor speed that is assigned to it, `processingDuration` is usually a function of these properties.

Figure 7.17 shows how a typical implementation for a resample task may look. This implementation models a situation in which the speed of resampling depends on the number of pixels of the largest image, which is the input image when scaling down, or the output image when scaling up. Moreover, on average, 20 clock cycles are used per pixel in the largest image. With this information, the `processingDuration` of the step can be computed. The implementation can directly refer to all properties of the associated task (such as the input pin and output pin properties).

```
implementation Resample_CPU performs Resample on CPU
{
  // We assume that a resample computation costs 20 clock
  // ticks per pixel, with the biggest of the two images
  // (depending on whether we zoom in or out) determining
  // the number of pixels

  cyclesPerPixel = 20;

  processingDuration =
    max(out.size, in.size) * cyclesPerPixel * processor.assignedSpeed;
}
```

Fig. 7.17 An implementation.

A **priority**, formulated as an integer number, specifies which step gets to "go first" if multiple steps want to use a resource at the same time. We enforce one important convention with respect to priorities: the higher the number, the lower the priority. So a step with priority 3 is considered more important than a step with priority 7. It is possible for multiple execution links to have the same priority, and this should imply that resources are fairly shared between the competing steps.

7.5 DSEIR: DSE Intermediate Representation

Section 7.3 motivated the importance of an intermediate representation to support DSE. We are developing the intermediate representation DSEIR specifically for the purpose of model-driven DSE. In modelling design alternatives, DSEIR follows the Y-chart paradigm. It further has support for defining experiments. It has been realised as a Java library, filling in the central module of the architecture of Fig. 7.4 on page 5. The current implementation supports four views: application, platform, mapping, and experiment. DSEIR can be used through a Java interface, an XML interface, and an Eclipse-based prototype GUI. This section introduces and illustrates the four views of DSEIR. We use the DSE case study of the previous section as a running example.

7.5.1 DSEIR: The Application View

Figure 7.18 shows a fragment of the DSEIR representation of the application part of our running printer example, in both the XML and the graphical format. The DSEIR application language is inspired by dataflow languages, in which data transformations play the most prominent role, but it intends to also support Petri-net and automata-based modelling concepts. An application consists of a number of tasks, and models the functional behaviour of the system. Each task has one or more ports, a number of load declarations and a number of edges. Ports are collections of values of some type (integer, integer array) and are either ordered in a fifo (first-in first-out) way or unordered. Ports provide inputs to tasks. The load declarations specify the load of the task for the *services* that it uses. These services are expected to be provided by the platform (such as a COMPUTATION service that is provided by a CPU). The task loads are used in combination with platform information to determine the execution time of the task. The use of *service types* avoids direct references to platform resources which avoids coupling of the application description with a specific platform description. An edge leads from the current task to either a port of another task or to a port of the same task. The purpose of an edge is to add a new value to the target port. An edge has an *expression* in the DSEIR expression language that gives the new value in the target port. Furthermore, both a task and an edge can have a *condition*, which is a boolean expression that determines whether the task or edge is enabled and can execute. Finally, an edge can have zero or more *handover* specifications (the '`ho`' entries in the XML representation in Fig. 7.18) which contain an amount of allocated services that should be passed on to the next task. This allows modelling of resource reservations, for instance, memory pointers that are passed on from one task to the other without releasing the memory. An application can have a number of global variables of type integer, and a number of parameters, which also are of type integer. Both can be used in expressions in the application part. The DSEIR expression language is sufficiently powerful to capture applications at the

(a)

```
<param name="NumPages" />

<global name="TIME_SCALE" value="1000" />
<global name="SCAN_CF" value="10" />
<global name="IP1_LOAD" value="300" />
<global name="A4_PIXELS_600_DPI" value="34802530" />
<global name="A4_RGB_BYTES_600_DPI" value="3*A4_PIXELS_600_DPI" />
<global name="NumBands" value="(A4_RGB_BYTES_600_DPI / (1024 * 1024 * 10)) + 1" />

<task id="Download">
    <port id="p" type="int" init="1" />
    <load service="TRANSFER" value="A4_RGB_BYTES_600_DPI/(SCAN_CF*1024)" />
    <load service="RESULT_STORAGE" value="1" />
    <edge port="p" value="p+1" cnd="p &lt; NumPages" />
    <edge port="p_b" task="IP1" value="[p,1]">
        <ho service="RESULT_STORAGE" value="1" />
    </edge>
</task>

<task id="IP1" cnd="p_b[0] == next_p">
    <port id="p_b" type="int[2]" order="unordered" />
    <port id="next_p" type="int" init="1" />
    <load service="COMPUTATION" value="(IP1_LOAD*TIME_SCALE)/NumBands)" />
    <load service="INTERNAL_STORAGE" value="1" />
    <load service="RESULT_STORAGE" value="1" />
    <edge port="p_b" value="[p_b[0], p_b[1]+1]" cnd="p_b[1] &lt; NumBands">
        <ho service="INTERNAL_STORAGE" value="1" />
    </edge>
    <edge port="next_p" value="if (p_b[1] &lt; NumBands) then next_p else next_p+1" />
    <edge task="IP2" port="p_b" value="p_b">
        <ho service="RESULT_STORAGE" value="1" />
    </edge>
</task>
```

(b)

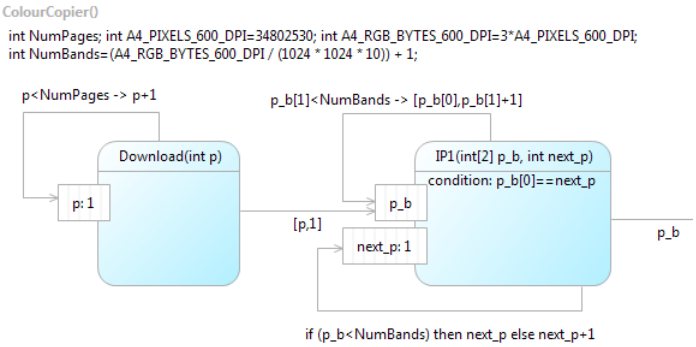

Fig. 7.18 An application in DSEIR: (a) XML; (b) graphical representation.

intended abstraction level; it is kept as simple as possible though to facilitate model transformations to analysis tools.

Figure 7.18 shows the specifications for the Download and IP1 tasks of the running example. The XML representation includes the load and handover specifications, which in the figure are omitted from the graphical representation. Comparing the DPML model of the previous section with the DSEIR model, we see some differences. First of all, all concepts in DSEIR are independent of any specific application domain, whereas DPML intentionally contains elements from the domain of professional printing (with built-in concepts like pages, `PixelData` pins, and printer and scanner resources). Furthermore, the conversion from pages to bands is

explicitly visible in the DSEIR task graph, whereas it is part of the implementation specifications in DPML. The most important difference, however, is the fact that DSEIR allows to specify task workloads and high-level resource management aspects in an abstract way in the application view, via services, loads, and handovers. This allows a strict decoupling between application and platform aspects, as further illustrated below. This fits with the goal of DSEIR as a domain-independent intermediate representation, which should allow to capture a wide diversity of mapping, scheduling, and resource allocation strategies. For a domain-specific language like DPML, predefined solutions for some of these aspects may be acceptable, which keeps the language simpler and more intuitive for domain engineers.

The semantics of a DSEIR application model is Petri-net like. A task can execute if all its ports have at least one value and if the condition of the task evaluates to *true* for the chosen port values. A task can choose any value from an unordered port or the first value from a fifo port. Upon execution start it consumes the chosen value from each port. When the task is finished, it executes all its enabled actions in an atomic fashion, which produces new values in a (sub)set of the target ports.

```
<resource id="cpu" capacity="4">
    <time service="COMPUTATION" value="1" />
</resource>

<resource id="ethernet" capacity="1">
    <time service="TRANSFER" value="(1000 * TIME_SCALE)
</resource>

<resource id="b1" capacity="b1_size">
    <time service="INTERNAL_STORAGE" value="0" />
    <time service="RESULT_STORAGE" value="0" />
</resource>

<resource id="b2" capacity="b2_size">
    <time service="INTERNAL_STORAGE" value="0" />
    <time service="RESULT_STORAGE" value="0" />
</resource>
```

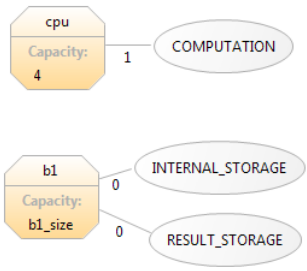

Fig. 7.19 DSEIR resource definitions in XML and graphical format.

7.5.2 DSEIR: The Platform View

A platform in DSEIR consists of a number of resource declarations. Figure 7.19 shows a fragment of the DSEIR platform representation for our running printer example. Each resource has a capacity which can be read as the number of available units. Furthermore, a resource provides a number of *services* that tasks can use. A resource has a certain *service time* for each service it provides. This service time equals the number of time units that is needed to process one unit of load. This is used in combination with the load of a task to compute the task's execution time. The platform in the example has a quad-core CPU resource that provides a COMPUTATION service; the buffers provide INTERNAL_STORAGE and RESULT_STORAGE

services, which allows to distinguish buffers for input and output data. The service time of the buffers is set to 0, which corresponds to an infinite processing speed. A platform can also have a number of parameters, which, like the application parameters, are of type integer. These can be used in expressions in the platform part, such as the capacity and speed expressions of resources. In the example, the buffer sizes are parameters (to be optimised in the DSE).

In line with the motivation for application-centric modelling laid out in Sect. 7.3 and in contrast to DPML models, a DSEIR platform model is simply a collection of resources without explicit structure. The structured platform models of DPML conform to the typical views of designers, whereas the unstructured models of DSEIR fit well with an intermediate representation that should be conceptually as simple as possible.

7.5.3 DSEIR: The Mapping View

The mapping ties an application to a platform, and consists of an *allocator* entry for each task, and *priority* and *deadline* specifications. An allocator element specifies to which resources the services required by the application are mapped. Furthermore, it specifies the amount of the resource that is allocated to the task. This amount should be less than or equal to the resource capacity. An allocator can be preemptive; by default it is non-preemptive. If it is preemptive then a running task can be preempted and the preemptive resource can be allocated to another task. If an allocator is non-preemptive and the available resource capacity is not enough for the task, then the task cannot run. Such resource-arbitration choices are made at time 0 (the start of system execution) and each time a task finishes. The priority elements specify the priority of tasks. By default, tasks have priority 0 (the lowest priority in DSEIR). The priority must be an integer expression and can depend on the run-time state, e.g. the port values for the current execution of the task. This allows full dynamic priority scheduling. Deadline specifications can be used for schedulability analysis (see [40]).

Figure 7.20 shows fragments of a DSEIR mapping representation. The visual representation allows to reuse allocators for multiple tasks. In the running example, however, each task has its own allocator, with the same name. This is because in this example each task asks for a unique combination of resources. Task IP1, for instance, is bound to allocator IP1, which provides INTERNAL_STORAGE through buffer b1, RESULT_STORAGE through buffer b2, and COMPUTATION through the CPU resource. Allocator IP2 (not shown in Fig. 7.20) binds buffer b2 for INTERNAL_STORAGE to task IP2, thus, accurately capturing the sharing of b2. Priorities are left unspecified, resulting in the default (lowest) priority of 0 for all tasks. Deadlines are also left unspecified, because they are not used in this example.

The examples given throughout this section show how the intended Y-chart separation of concerns between application and platform is achieved through a mapping view. The only interaction is through service definitions and allocators. Application

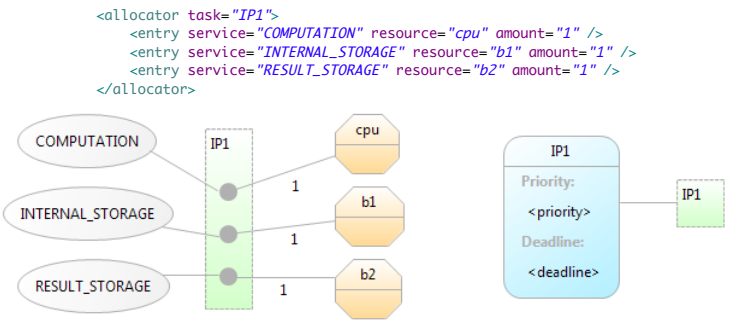

Fig. 7.20 Mapping in DSEIR.

and platform definition can be adapted fully transparently, as long as the service names do not change. The execution model of DSEIR is based on dynamic priority-based preemptive scheduling, which is a generic mechanism that allows designers to specify their own resource allocation and scheduling strategies through the allocator definitions. This fits with the needs of a generic intermediate representation. Considering the domain-specific language DPML, the resource allocation and scheduling mechanism is fixed in the native simulator, which on the one hand limits flexibility but on the other hand relieves designers from the task of specifying these aspects.

7.5.4 DSEIR: The Experiment View

DSE involves more than specifying design alternatives. It also requires the definition of experiments, among others. Experiments can also be described in DSEIR. Figure 7.21 shows an experiment definition. An experiment definition contains one or more experiment entries. If there is more than one, then these experiments are executed sequentially. An experiment entry has a name, that identifies the type of experiment, and can contain a *model* entry. The model entry can specify *one or more* models. Multiple models are specified using model parameters that may take different values. Finally, an experiment contains a number of properties. Every experiment type has its own set of properties with their own meaning. These experiment entries can be seen as invocations of predefined analysis recipes.

The example in Fig. 7.21 takes the models for our running example. The first experiment entry analyses all possible buffer size combinations for the buffers allocated in the main RAM memory, for a range of compression factors (parameters `minCF` and `maxCF`) and for 100 pages. It performs simulations (using CPN Tools [34], see next section) to explore the throughput (defined by the '`observers`' entry) that can be achieved for each combination of buffer sizes. Per combination, 10 simulations are performed (defined by the '`number`' entry). The other parameters set some values to format the output of the simulations. The

```
<experiment name="generate_traces">
    <model mapping="mapping.xml">
        <pvalue name="NumPages" values="100" />
        <pvalue name="minCF" values="10" />
        <pvalue name="maxCF" values="25" />
        <pvalue name="b1_size" values="1,2,3,4" />
        <pvalue name="b2_size" values="1,2,3,4" />
        <pvalue name="b4_size" values="1,2,3,4" />
    </model>
    <property name="observers" value="rt(sink)" />
    <property name="outputDir" value="traces" />
    <property name="number" value="10" />
    <property name="timeScaleDivision" value="1000000.0" />
    <property name="objectName" value="p" />
    <property name="launchResVis" value="false" />
</experiment>

<experiment name="extract_paralyzer_view">
    <property name="sourceDir" value="traces" />
    <property name="quantities" value="Timeunits per image = rt,
                RAM = b1_size * 25 + b2_size * 10 + b4_size * 6.2" />
    <property name="launchParalyzer" value="true" />
</experiment>
```

Fig. 7.21 An experiment in DSEIR.

second experiment entry takes the output and extracts a Pareto space that illustrates the trade-offs in the space, taking into account the variations that occur due to variation in compression factors. More details about the latter are given in the section presenting the diagnostic support in the Octopus tool set, Sect. 7.7.

The experiment view is an important part of DSEIR that allows designers to specify and maintain experiments. It is, for example, straightforward to re-run the same experiment on variants of a model. The tool set implementation, explained in some detail in Sect. 7.8, is such that it is easy to add new analysis plugins. An analysis plugin predefines an analysis recipe, as mentioned above, defining which tools are called, in which order, and with which parameter settings. This allows for example to easily add domain-specific analyses.

7.6 Analysis and Model Transformations

The previous two sections have introduced DPML, which served as an illustration of domain support that can be provided, and the intermediate representation, DSEIR. Together, DPML and DSEIR fill in the modelling perspective in our model-driven DSE philosophy and in the Octopus tool set. Design alternatives can be captured and it is possible to define experiments to explore the space of alternatives. The experiments may perform various types of analysis on the specified design alternatives. This section presents *three types of analysis* supported in Octopus.

From an industrial perspective, *simulation* is the most important analysis technique. Simulation technology is mature and it may serve many different purposes, ranging from building a basic understanding of system behaviour, to detailed timing analysis and functional validation. The current tool set uses CPN Tools [34] for simulation of DSEIR models.

Another class of widely used analysis techniques are *model checking* techniques that are based on the underlying principle that a model is exhaustively analysed to conclude whether or not properties of interest hold for the model at hand. Octopus supports translation to the Uppaal [10] model checker. Uppaal offers (timed and untimed) model checking, which may be used to perform deadlock analysis, property checking, and timing and schedule optimisation.

Finally, for data-intensive applications, like print pipelines, performance and resource usage are often dominated by the flow of data through the system and the operations performed on these data; this is in contrast to control-intensive operations where communication and synchronisation typically determine the performance. Specialised *dataflow analysis* techniques allow fast exploration of design alternatives at a high level of abstraction. Octopus supports an experimental interface to the SDF3 [62] analysis tool for dataflow analysis.

7.6.1 Simulation with CPN Tools

Coloured Petri nets (CPNs) [35] are an expressive, precisely defined, and well-established formalism, extending classical Petri nets with data, time, and hierarchy. CPNs have been used in many domains (e.g. manufacturing, workflow management, distributed computing, and embedded systems). CPN Tools provides a powerful framework for modelling CPNs and for performance analysis (stochastic discrete-event simulation) on CPN models.

DSEIR as outlined in the previous section defines a syntax. An earlier version of DSEIR has a precisely defined semantics [70], defined by means of a structural operational semantics. This semantics elegantly separates the Y-chart aspects (application, platform, and mapping). It would be possible to provide also the current version with a semantics along these lines. However, since DSEIR also needs execution support, we have chosen to provide both the semantics and the execution support via a transformation to CPNs. The goal of the transformation from DSEIR to CPN Tools in the Octopus tool set is therefore twofold, namely (1) to precisely define the semantics of DSEIR, and (2) to provide execution support for the full DSEIR language.

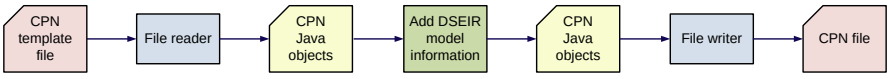

Fig. 7.22 Translating DSEIR specifications to CPN models.

Figure 7.22 illustrates the setup of the transformation from DSEIR models to CPNs. The basis of the transformation is a CPN template that contains the basic structure of the CPN model to be generated, the high-level dynamics of the resource handling, and monitors for producing simulation output. The template is filled with

the information from a concrete DSEIR model. The resulting CPN model can then be simulated by CPN Tools. Currently, there is an analysis recipe (see Sect. 7.8) that allows to simulate a specified number of runs of a given model. The execution traces resulting from these runs can then be further analysed, extracting properties such as the average throughput or resource utilisation, or observed bounds on performance properties such as latency or resource usage.

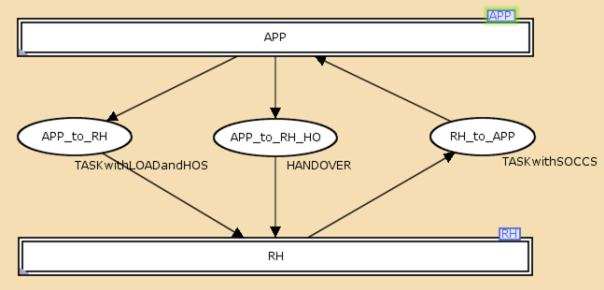

Fig. 7.23 Top-level view of the CPN model generated for the running example.

Figure 7.23 shows the top-level view of the CPN model generated for our running example. It illustrates the main structure of the CPN template used in the translation from DSEIR to CPN Tools. Any generated model consists of (1) the application view (block *APP*; a hierarchical transition in CPN terminology), (2) the specification of the resource handler (hierarchical transition *RH*), and (3) the interface between the two (components *APP_TO_RH*, *APP_TO_RH_HO*, and *RH_TO_APP*, called places in CPN terms). The information going from the application model to the resource handler contains the definition of a task to be started, the initial load of this task, and the handovers the task expects to receive. Information of the actual resource amounts (in terms of services) that a task occupies and notifications of a task being finished are communicated from resource handler to application.

Figure 7.24 shows the translation of the Download task, which is part of the application model generated for our running example. The task is split into a start event (transition *Download_s*) and an end event (transition *Download_e*). The first event sends task information to the resource handler (with the initial load specified in Fig. 7.18(a)) and puts a token into the waiting place *p_SE_Download*; the second event occurs when the resource handler informs the application layer that the task has been finished. When this happens, the complete result storage is sent as a handover to the next task (see the annotation of the arc to place *to_RH_HO*) and an incremented page number is sent to place *p_Download_in_port_p*; the latter represents the self-loop of the Download task in Fig. 7.18(b)).

Figure 7.25 shows the internals of the resource handler generated for the running example; we briefly explain the logic of the resource-handling mechanism. Transition *UpdateHandover* ensures that arriving handovers are properly processed; this transition has the highest priority (500). Transition *rcv_tran* accepts new tasks and

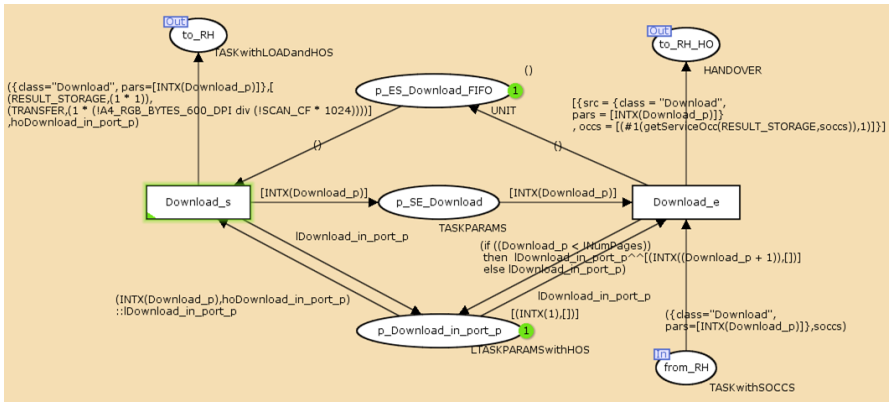

Fig. 7.24 A fragment of the generated CPN application model for the running example.

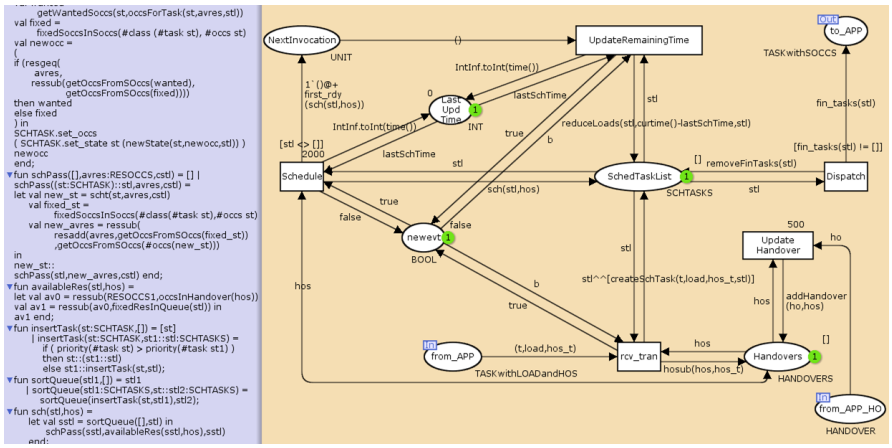

Fig. 7.25 The resource handler for the running example.

adds them into the queue of running tasks (place *SchedTaskList*). Transition *Dispatch* removes finished tasks from the queue and informs the application layer. The actual scheduling is done by the *Schedule* transition which has the lowest priority (2000) and executes only if the queue is not empty. The function *sch* (shown on the left) modifies the task queue, according to the rules defined by the mapping specification. Place *NextInvocation* ensures that time progresses only to the first moment when some running task gets done. Transition *UpdateRemainingTime* updates the load of running tasks to reflect progress of time. Places *newevt* and *LastUpdTime* are auxiliary places to ensure a correct ordering of transitions. Note that the major part of the dynamics of the resource handler is premodelled and stored in the template. Only bodies of already specified functions are filled in when generating a CPN model for a concrete DSEIR specification.

The translation from DSEIR to CPN Tools is fast; also the simulations themselves are fast and scalable. However, CPN Tools compiles a model into an executable for performing the simulations. This compilation step is the slowest part in the transformation and analysis trajectory. For the exploration of large design spaces, in which many alternative models are simulated, this compilation step may become very time consuming.

7.6.2 Analysis with Uppaal

The timed automata formalism extends traditional finite-state automata with real-valued clocks [4]. This results in a concise formalism that is well suited to model state-based systems in which time plays a role. Properties of interest for such models can be phrased in temporal logic. Timed Computation Tree Logic (TCTL) is a logic that allows to specify properties with respect to the reachability of states within specified time bounds. This allows to specify, for example, that a page should be processed within a given latency, or that a certain error state should not be reachable. The fact that TCTL is decidable for timed automata [2] has led to the development of a number of analysis tools, *model checkers*, that compute whether a given timed automaton model satisfies a given TCTL specification. This section discusses the link from DSEIR to the Uppaal [10] model checker. The Uppaal input language extends the lean timed automata formalism with data (integer variables, language constructs to create C-like structures, etc.) and a C-like language to manipulate data. These features ease the creation and maintenance of models.

The translation from DSEIR to Uppaal is based on the following principles. First, every task in the application is translated to a separate Uppaal timed automaton, which has a clock to track the progress of the task. Tasks communicate through global variables that model the ports of the tasks. Second, resources are also modelled by global variables. Third, tasks read and write these in order to implement the allocation strategies as defined in DSEIR.

Figure 7.26 shows the Uppaal timed automaton for the Download task of the running example. The transition from *initialize* to *idle* initialises the port of the Download task with its initial value. The transition from *idle* to *active* models the start of the task. It picks the first port value (index $i0$) because the port is fifo. Furthermore, functions such as *claim* are called in order to do the bookkeeping with respect to resources, and the clock x is set to 0. The transition from *active* to *idle* models the completion of the task. The *release* function releases the resources and *produce* generates the values for the tasks' outgoing edges.

The main strength of the Uppaal model checker is that it enables *exhaustive* analysis of a DSEIR model. Currently, there are analysis recipes (see Sect. 7.8) to (1) check for deadlock situations, and (2) find precise bounds on resource usage (used memory and queue sizes, for instance) and latency. In addition, an experimental version of Uppaal-based schedulability analysis has been implemented (see [40]). These analysis recipes are only applicable to small to medium-sized models

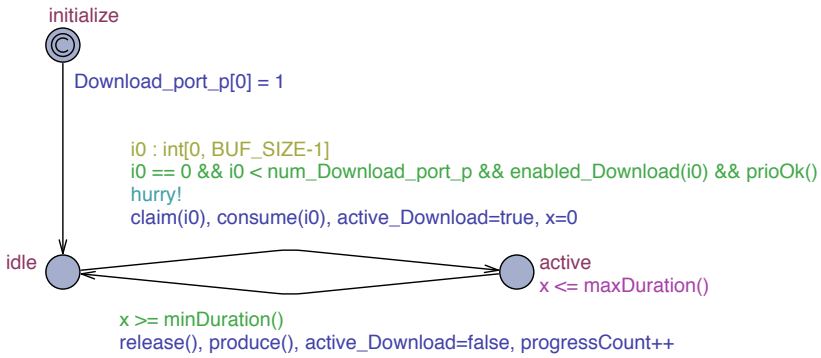

Fig. 7.26 The Uppaal timed automaton of the Download task from the running example.

with limited non-determinism, because of the state-space explosion that is inherent in model checking. State-space explosion refers to the exponential growth of the state space with increasing model size. In this respect, model checking techniques contrast with simulation-based techniques which scale much better (but are not exhaustive).

Not all aspects of the DSEIR language are translatable to Uppaal. The main concept that cannot be translated directly is the concept of preemption. The DSEIR language is targeted at the system level of software-intensive embedded systems. This motivated the choice to allow DSEIR to approximate the progress of task execution by piece-wise linear behaviour. Consider, for instance, the situation that two tasks share the same processor core. The fine-grained division of time that may occur in reality is approximated by slowing both tasks down by some factor. This cannot be modelled by timed automata, although an approximation has been presented in [29, 26]. This approximation, however, fragments the symbolic state space built by Uppaal, which has a strong negative effect on the scalability of Uppaal analyses.

Two extensions of the Uppaal tool provide means to deal with preemptive behaviour more elegantly. First, Uppaal supports analysis of stopwatch automata (timed automata in which clocks can be stopped [14]) based on an over-approximation. This is useful to model situations in which a task is completely preempted. This type of preemption is actually what is covered by the current translation from DSEIR to Uppaal; the aforementioned approximation is not supported. Second, the Statistical Model Checking (SMC) extension of Uppaal [18, 13] features networks of priced timed automata, where clocks may have different rates in different locations. These networks of priced timed automata are as expressive as general linear hybrid automata [3]. SMC essentially provides stochastic discrete-event simulation for the combination of the timed automata and TCTL formalisms. The linear hybrid automata formalism is ideal for expressing the piece-wise linear progress of tasks as described above. The translation to this Uppaal variant, however, has not yet been realised because Uppaal-SMC has only been developed recently.

Besides the fundamental limitation with respect to preemptive behaviour, there are some practical limitations that have to do with the present Uppaal implementation: (1) the limited range of variables (32 bits for integers and 16 bits for clocks) sometimes leads to inaccuracies in approximating real values and to scaling problems in terms of the length and duration of executions being analysed; (2) the model state in Uppaal needs to be statically defined, which does not match well with the fact that DSEIR models do not have a priori bounds on the state (the port contents); this may lead to run-time errors that are typically hard to diagnose.

Supporting model transformations from DSEIR to multiple analysis tools raises the interesting question of consistency between these transformations. One may pick up the challenge to formally prove the correctness of a transformation with respect to the DSEIR semantics. An illustration of the type of proofs needed to show correctness of transformations can be found in [26, 71]. Those proofs were done in the context of an earlier version of DSEIR. We did not provide a formal correctness proof of the translation to Uppaal with respect to the CPN semantics for the current version of DSEIR. Pragmatically, when applying the CPN Tools and Uppaal analyses on models specified in the common subset of DSEIR supported by both translations, we get the same outcome.

7.6.3 Dataflow Analysis with SDF3

The model transformations discussed in the previous two subsections provide simulation support for the full DSEIR language and specialised analysis support for a subset of the language. The transformation from DSEIR to dataflow is of a different nature. The target of the transformation is Resource-Aware Synchronous DataFlow (RASDF) [77, 76], which extends the classical Synchronous DataFlow (SDF) [41] model of computation with a notion of resources in the style of the Y-chart. SDF has limited expressiveness, but this restriction allows more powerful analysis. It is for example possible to minimise memory requirements for a given throughput requirement [60, 78].

SDF models tasks by means of actors with a fixed execution time and tasks are assumed to communicate fixed amounts of data in all their executions. SDF therefore does not allow to capture dynamics such as variable execution times and communication rates explicitly. With conservative (worst-case) task execution times and communication rates, however, it is possible to determine bounds on performance and resource usage, such as the minimal throughput that can be guaranteed and the smallest amount of memory that suffices to guarantee that throughput. The model transformation from DSEIR to RASDF therefore aims to translate a subset of the DSEIR language to RASDF models that are conservative in terms of performance and resource usage. Figure 7.27 illustrates the transformation.

The figure shows that it is possible to directly transform a restricted subset of DSEIR, DSEIR-RASDF, to RASDF. DSEIR-RASDF is the subset that simply limits DSEIR to RASDF models. Using static analysis of RASDF models, it is possible

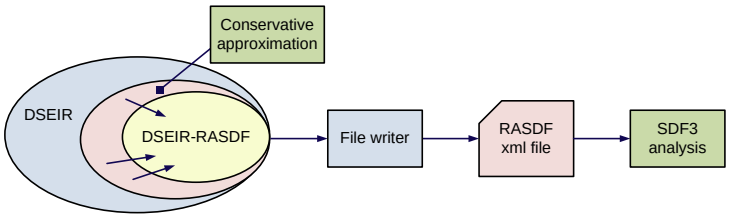

Fig. 7.27 The transformation from DSEIR to RASDF/SDF3.

to enlarge the subset of DSEIR models that can be conservatively captured as a DSEIR-RASDF model. DSEIR-RASDF models can then be exported to the SDF3 tool [62] for analysis, allowing the already mentioned throughput and resource usage analysis. More details about this transformation can be found in [48].

The described approach can only be applied to models with limited variation in task execution times and communication rates. If the variation in execution times and communication rates is too large, such as for example the variations because of compression rates in the running example (see Sect. 7.7), then the conservative bounds on the throughput and memory requirements that can be guaranteed are too loose to be practically useful. As a future extension, it will be interesting to investigate a link to Scenario-Aware DataFlow (SADF) [67, 79]. SADF allows to capture a finite, discrete number of workload scenarios, each characterised by an SDF model. A workload scenario may correspond to for example different types of pages (text, image) to be printed. Scenario transitions can then be captured in a state-based model such as a finite state machine or a Markov chain. Many analysis techniques for SDF can be generalised to SADF [61], which therefore provides an interesting compromise between expressiveness and analysis opportunities.

7.7 Diagnostics

The Octopus tool set has two tools for visualisation and diagnostics. The ResVis tool, short for *Res*ource *Vis*ualisation, see [59], can be used for detailed analysis of the behaviour of individual design alternatives. The Paralyzer tool, short for *Par*eto an*alyzer*, see [23], on the other hand, provides Pareto analysis and supports trade-off analysis between different design alternatives. The tools thus are complementary to each other. They fill in the diagnostics module of Fig. 7.4 on page 5.

7.7.1 Visualisation and Analysis of Execution Traces

The ResVis tool can be used to visualise individual executions of the modelled system by means of a Gantt chart. A Gantt chart shows the task activity and/or resource

usage over time. Figure 7.2 on page 4 shows a ResVis Gantt chart for part of an event trace from one of the case studies performed with Océ-Technologies. Figure 7.28 shows a part of an event trace (top) for the running colour copier example and the accompanying resource plots of buffers b2 and b4 (middle and bottom, respectively). The model for the running example reserves one slot for b1 and four slots for b2 and b4. The event traces show the execution of the individual tasks. The resource plots show the resource usage of individual resources. The tasks and resource usage blocks can be coloured according to one of some predefined schemes such as by page number, use case (e.g. printing, scanning), or job (e.g. printing a specific number of pages of a given type). This is specific for the printer domain. The colouring in Figs. 7.2 and 7.28 is by page number.

Fig. 7.28 Gantt chart and resource plots of buffers b2 and b4 of the running example.

Event traces and resource plots are very suitable to study the detailed dynamic behaviour of a design. They show for example implicit dependencies between tasks such as unexpected blocking and utilisation of resources. A typical use is to find bottleneck tasks and bottleneck resources. These are tasks and resources that determine the performance of the system. The Octopus tool set contains algorithms to compute an over-approximation of the set of critical tasks to support this type of analysis [24].

For instance, Fig. 7.29 shows the critical tasks of the trace shown in Fig. 7.28. Visualisation of critical tasks can ease the identification of bottleneck tasks and re-

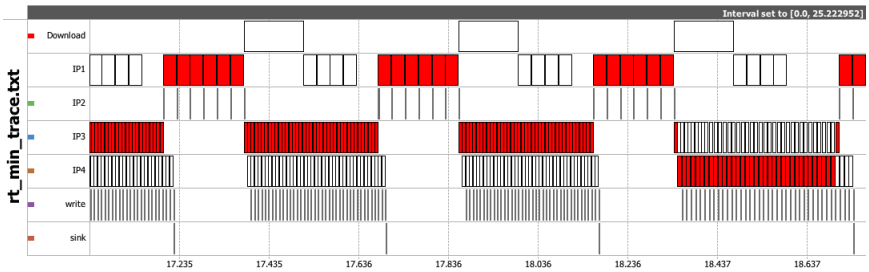

Fig. 7.29 The critical tasks in the execution trace of Fig. 7.28 are highlighted.

sources. Figure 7.29 shows that IP1, IP2, IP3, and IP4 form the critical path. (The colouring is not visible for the very fast task IP2.) Note that the application as shown in Fig. 7.9 has no data dependencies from IP3 to IP2 and also not from IP2 to IP1. Yet, the critical tasks show that IP1 must sometimes wait for IP2, and that IP2 must sometimes wait for IP3. This is caused by full buffers between these tasks, which limits parallelism.

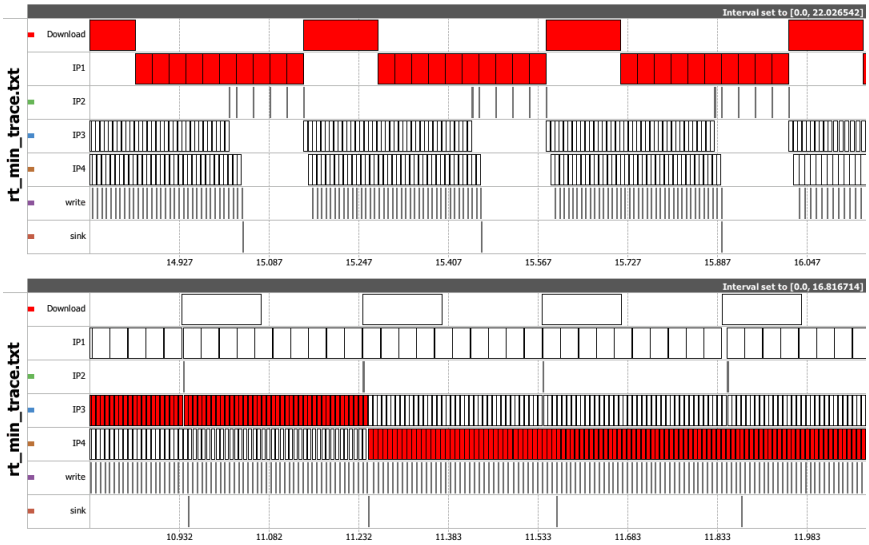

Fig. 7.30 The critical tasks after increasing the number of slots of b2 to 10 (above), and after an additional increase of the number of slots of b1 to 2 (below).

Figure 7.30 shows critical tasks after an increase of b2 to 10 (top graph) and after an additional increase of b1 to 2 (bottom graph). Both changes positively affect the throughput of the system. After the last increase, there is little or no room left

for further improvement. The critical path visualisation is a great help for solving problems with respect to time-related performance issues.

7.7.2 Trade-Off Analysis with Uncertain Information

Design-space exploration is a multi-objective optimisation problem. On the one hand, there is the design space consisting of all possible design alternatives. On the other hand, there is the cost space with its multiple cost dimensions, such as throughput, total memory usage, etc. Every design alternative is linked to the cost space, and the question is to find the best design solutions with respect to the considered cost dimensions. Pareto analysis is a well-known way to deal with this [53]. The Octopus tool set uses the Paralyzer library to perform Pareto analysis, including the application of constraints and cost functions, and a means to visualise the trade-offs in two or more dimensions [23]. Furthermore, it can cope with uncertainty, which is typically present in the early phases of system design, by associating a design alternative with *sets* of points in the cost space, not with just a single point.

Consider the running example introduced in Sect. 7.4.1. The DSE question asked is: Minimise the amount of memory allocated to the buffers while retaining a minimum given throughput. In order to answer this question, a DSEIR model was created, which is parameterised with the three buffer sizes; fragments of this model are shown in Sect. 7.5. The possible values for the buffer sizes are taken from the set $\{1,2,\ldots,10\}$, which gives a design space of $10 \times 10 \times 10 = 1000$ design alternatives. The Octopus tool set has been used to compute the throughput and the memory consumption of this set of design alternatives. The exact analysis through the Uppaal model checker and the analysis recipe for analysing bounds as mentioned in the previous section turns out to be too slow and the state space becomes too large because of the workload variations (different compression factors) and the number of pages in a job. Therefore, 30 simulation runs per configuration were made. The results are shown in Fig. 7.31.

The graph shows the cost space of the DSE question. The y-axis shows memory used for buffering; the x-axis shows throughput. The inverse of throughput is used so less is better. Each coloured rectangle represents a Pareto-optimal design alternative. This is a design alternative which is not dominated by any other design alternative, where a design alternative dominates another one if it is not worse in any dimension of the cost space and better in at least one dimension. For instance, the blue rectangle at the right-hand side represents the design alternative in which all buffers have size 1. This is the cheapest design alternative in terms of memory usage, but its throughput is low. Note that design alternatives are associated with a subset of the cost space and not with a single point. Each design alternative has variation in its throughput caused by variations (stochastic behaviour) in the input (not every image is the same, leading to different compression rates). This variation is visualised by using rectangles of various sizes instead of points in the graph. Because of the use of simulation, the throughput bounds for each of the rectangles

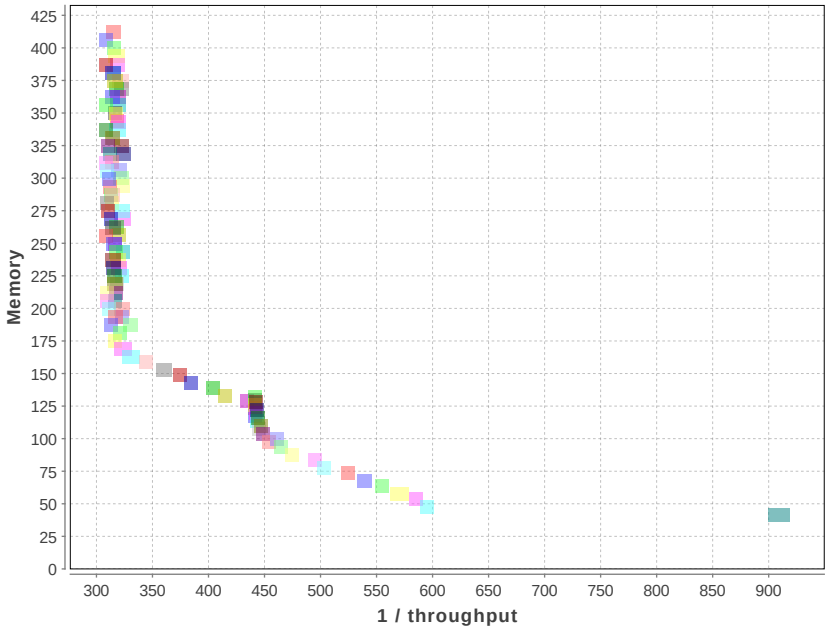

Fig. 7.31 Trade-off view of the total buffer size versus the throughput for the running example.

are only approximations of the true bounds. The graph shows that the throughput increases with larger buffers. However, using more than 175 units of memory does not result in a significant further increase in performance. The many configurations with more than 175 units of memory are on the Pareto front because they overlap in the throughput dimension. They are therefore not dominated by other configurations. This is a consequence of the fact that design alternatives are associated with sets of points in the cost space instead of with a single point.

7.8 Implementation Aspects

The Octopus tool set currently consists of various separate applications: the domain-specific data path tooling introduced in Sect. 7.4, the ResVis and Paralyzer diagnostics tools, which have been discussed in the previous section, and the generic Octopus application which is discussed in this section. The main aim of the Octopus application is to provide a formal, flexible, and extensible infrastructure to efficiently solve DSE problems that have been modelled in the DSEIR modelling language described in Sect. 7.5.

A DSE problem, by definition, has a number of design alternatives that need to be analysed. The analysis runs for different design alternatives are typically inde-

pendent (although it is an interesting topic for further research to investigate incremental DSE techniques where analysis results for one design alternative can be reused for other alternatives). Evaluation of design alternatives can be distributed with little effort over a possibly heterogeneous set of computational nodes. The Octopus implementation has built-in support to automatically distribute analyses over computational nodes and collect results from these analyses. Almost any modern computer has multiple processing cores, so this support is very useful in practice. The simulations performed for the running example of the high-end colour copier reported on in the previous section show that distribution indeed can be very effective. The computation time decreases almost linearly with the number of available computational nodes.

The extensibility requirement for the Octopus tool set, together with the wish for platform independence and distribution support have led to the decision to build Octopus upon the OSGi runtime [52] environment. This Java-based module framework facilitates extensible service-oriented architectures. An important feature is the service registry which enables publication and lookup of services (implementations of Java interfaces). The modules are so-called OSGi bundles, which are plain JAR files. The OSGi framework enables dynamic addition, update, and removal of bundles and of services. Typically, bundles use other services to implement their own service interface. This service orientation often results in loosely coupled and easily testable components. The OSGi implementation that is currently used is Apache Felix [5]. The Octopus workflow treats both models and analysis results as data that can be transformed and visualised. The Octopus implementation architecture and data-centric approach is similar to that of the macroscopic tools [12] and, more specific, the CIShell [16] as described by Börner.

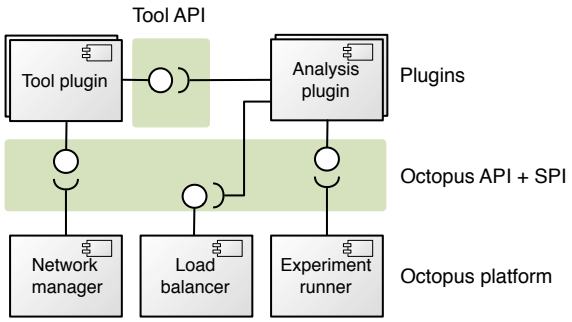

Fig. 7.32 High-level architectural overview of the Octopus implementation.

Figure 7.32 shows the high-level architecture of the Octopus implementation. It consists of the following components:

- The *Octopus API* (application programming interface) consists of several parts. It contains types to create DSEIR models, either programmatically in Java or

from an XML description. Furthermore, it contains types for analysis results, such as a generic execution trace format. It also contains the service interfaces of the Octopus platform services that allow developers to program tool and analysis plugins, as discussed below.

- The *Octopus SPI* (service provider interface) contains the service interfaces that plugin developers should implement and register in the OSGi framework to extend the functionality of the tool set with a new analysis tool.
- The *Octopus platform* currently contains components that are necessary to realise the API and SPI functionality. Most notably, the *LoadBalancer* and *NetworkManager* realise the distribution capabilities of the tool set explained above, and the *ExperimentRunner* provides a user interface to run experiments.
- *Tool plugins* are OSGi bundles that register implementations for certain SPI types. They facilitate the use of dedicated analysis tools and typically implement a model transformation from DSEIR to the input language of the supported tool, and a transformation from the tool output to a general format specified in the Octopus API such as the trace format for execution traces. Tool plugins are free to specify their own API, as different tools can have very different functionality. Currently, as discussed in Sect. 7.6, there are mature plugins for CPN Tools [34] and Uppaal [10], and an experimental plugin for the dataflow analysis tool SDF3 [62].
- *Analysis plugins* provide a means for the user to use the tool plugins, and they thus typically use the APIs of the tool plugins and the Octopus API. These plugins form the analysis recipes that can be used in the DSEIR experiment view. They implement part of the SPI, in order to register themselves in the Octopus framework. The current version of Octopus has the following analysis plugins:
  - *GenerateTraces* uses the CPN Tools plugin to generate a specified number of execution traces for a given non-empty set of models, and collects the output in the form of execution traces. This plugin was used to generate the traces underlying the Pareto analysis shown in Fig. 7.31. Individual traces can be visualised by the ResVis diagnostics tool.
  - *VerifySystem* uses the Uppaal plugin to verify that the system satisfies some sanity properties such as, for example, that there is no deadlock.
  - *RandomUppaalTrace* uses the Uppaal plugin to generate a random trace. It uses a built-in Uppaal facility for generating execution traces, and makes this available to the Octopus user. In general, the GenerateTraces plugin is faster though.
  - *GenerateBounds* uses the Uppaal plugin to compute lower and upper bounds on application latency, resource usage, and port usage (the number of items present in any of the ports in a DSEIR application model).
  - *GenerateParalyzerView* collects performance data created by any of the aforementioned analysis plugins in order to generate a Pareto trade-off view as shown in Fig. 7.31.
  - *CriticalPathAnalyzer* applies critical-path analysis to a specific trace file. The results can then be visualised in ResVis, as illustrated in Figs. 7.29 and 7.30.

7.9 Industrial Experiences

We have used Octopus in four case studies at Océ-Technologies. These case studies all involve design-space exploration of printer data paths of professional printers.

7.9.1 High-Performance Production Black-and-White Printing

Figure 7.33 shows an abstracted view of an FPGA-based data path platform of a high-performance production black-and-white printer, that supports use cases such as printing, copying, scanning, scan-to-email, and print-from-store. All required image processing algorithms are realised in the main FPGA (the IP blocks in the figure); the FPGA is connected to two memories via a memory bus. The machine can be accessed locally through the scanner and both locally and remotely through a print controller. Print jobs enter the system through the data store shown in the figure. The use cases all use different combinations of components in the platform. A print job arriving from the Data Store undergoes several image processing steps, with intermediate results being stored in one of the memories, after which the processed result is both sent to the Printer block and stored in the Data Store. The latter is useful for error recovery and for printing multiple versions of the same document. A scan job uses the Scanner board and several IP blocks, with intermediate results stored in one of the memories and the final result stored in the Data Store. Scanning and printing can execute in parallel, and also within the scan and print image processing pipelines, tasks may be executed in parallel. Resources like the memory and the associated memory bus, as well as the USB are shared between tasks and between print and scan jobs running in parallel. Moreover, the available USB bandwidth dynamically fluctuates depending on whether it is used in one or in two directions simultaneously.

Given the characteristics of the use cases and the FPGA-based platform, the data path in this case study can be modelled quite accurately with fixed workloads (for the various processing tasks, the memory bus, and the memories) at the abstraction level of pages. Only the USB client shows variation due to the above-mentioned variation in available bandwidth between unidirectional and bidirectional use. Schedule optimisation using Uppaal analysis is therefore feasible. USB behaviour can either be approximated with a fixed bandwidth or with a discrete approximation of the fluctuating behaviour as explained in Sect. 7.6.2. The models we developed were used for determining performance bounds and resource bottlenecks, for analysing interaction between scanning and printing, and for exploration of scheduling priorities and resource allocation (memory allocation, page caching) alternatives. One of the concrete results was an improved task prioritisation for the static priority scheduling employed in the platform. Further, Fig. 7.34 shows the Gantt chart of simultaneously running scan (red) and print (blue) jobs. The scan job is disrupted, because printing has priority on the USB. It only continues after the print job has finished. This behaviour materialises because the model lacks a crucial

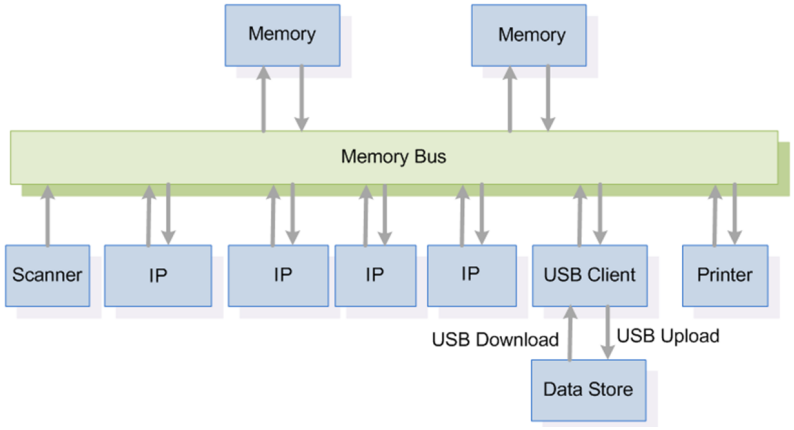

Fig. 7.33 An abstracted view of the platform of a high-performance production black-and-white print and scan data path. (Figure from [30])

scheduling rule, stating that uploads over the USB are handled in the order of arrival, irrespective of the origin (the scan or print job) of the upload. This scheduling rule is actually enforced in the print controller, which is a component external to the data path. The analysis shows the importance of this external scheduling rule.

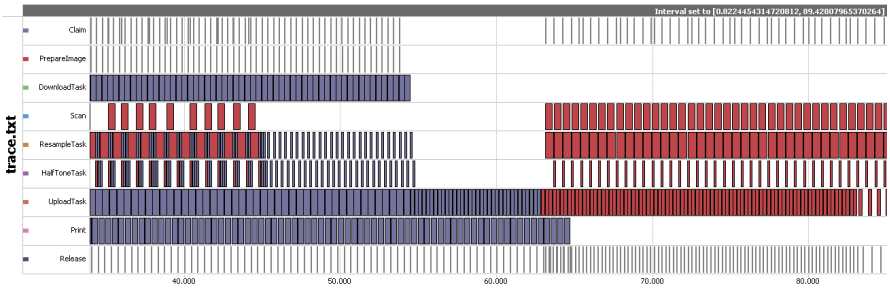

Fig. 7.34 The Gantt chart of simultaneously running scan jobs (red) and print jobs (blue).

An interesting conclusion with respect to the Octopus tool set is that automatically generated Uppaal models are in comparison better tractable than the handcrafted models reported on in [29]. Longer print jobs can be analysed, due to a reduction in memory needed by Uppaal. Analysis times increase though. Memory usage and analysis times strongly depend on the size of the print jobs being analysed. For latency optimisation of two simultaneously running jobs, memory usage and analysis time range from kilobytes and seconds for small jobs of a few pages to gigabytes and hours for large jobs with hundreds of pages. Details can be found in [26].

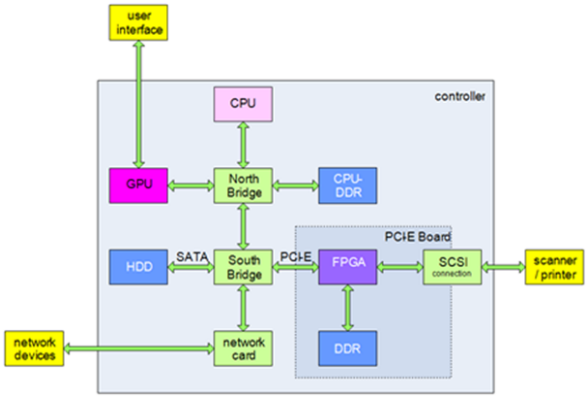

Fig. 7.35 A data path platform template used in several high-end colour copiers. (Figure from [8])

7.9.2 High-End Colour Printing

The other three case studies involved variants of a family of high-end colour copiers. Figure 7.35 shows a template of the data path platform used for these printers. It is a heterogeneous multi-processor platform that combines one or more CPUs (running Microsoft Windows) with a GPU, one or more Harddisks (HDDs), and an FPGA. Because of heterogeneity and the use of general CPUs, capturing the platform in a high-level abstraction is more challenging than modelling the platform of the first case study. The variation in workloads due to compression and decompression steps in the print and scan pipelines adds complexity, as well as the fact that pages are broken into bands and sometimes even lines to increase pipelining opportunities in the image processing pipeline. The latter is needed because high-resolution colour printing and scanning involves much larger volumes of data than black-and-white printing and scanning. The Gantt chart shown in Fig. 7.2 is in fact taken from one of these three case studies, and illustrative for the mentioned challenges. Also the running example is taken from one of these case studies.

The complexity of the models for the colour printer data path case studies is such that only simulation is sufficiently fast to do any practically meaningful analysis. For the first one of these cases, we started out with handcrafted models. Later we made DSEIR models, and for the last case study also DPML models. Automatically generated CPN models turned out to yield simulation times similar to the handcrafted CPN models. Simulation times range from seconds to minutes, depending on the size of the jobs being simulated. The translation of a DSEIR model to a CPN takes typically less than a second. The time that CPN Tools needs to compile a (handcrafted or generated) CPN model into a simulation executable is in the order of tens of seconds. Simulation times for DPML models with the native DPML simulator are of the same order of magnitude as CPN Tools simulation times.

Our analyses identified performance bounds for the print and scan pipelines and resource bottlenecks, and they were used to explore the interaction between

scanning and printing. Buffer requirements were analysed and potential savings in buffering were identified. The impact of several changes in the image processing pipelines were analysed before the changes were realised, and a sensitivity analysis was performed for task workloads that were not precisely known.

The three colour copier case studies showed that the Octopus tools can successfully deal with several modelling challenges, like heterogeneous processing platforms with CPUs, GPUs, FPGAs, and various buses, varying and mixed abstraction levels (pages, bands, lines), preemptive and non-preemptive scheduling, and stochastic workload variations (due to input variation and caching). The mixing of abstraction levels in a single model was crucial to obtain the appropriate combination of accuracy and acceptable simulation speed.

7.9.3 General Lessons Learned

We can draw some general conclusions from the performed case studies. DPML and DSEIR allow to capture industrially relevant DSE problems. Both DPML and DSEIR models can be made with little effort, similar to the time investment needed for a spreadsheet model. Because of the provided modelling templates, creating a DPML or DSEIR model takes much less effort than creating a CPN Tools or Uppaal model. An important advantage of the use of an intermediate representation is that one model suffices to use different analysis tools, which means a further, substantial reduction in modelling effort when compared to handcrafting models for multiple tools. Model consistency is moreover automatically guaranteed. The aspect that is in practice the most tedious and time-consuming part of the modelling are the task workload models (processing, bandwidth, and storage requirements). These workload models are typically estimates based on experience of engineers or based on profiling measurements on partial prototypes or on earlier, similar machines. Note that these workload models are typically independent of the chosen modelling approach. Spreadsheet models, DPML, DSEIR, CPN Tools, and Uppaal alike need the same workload models as input. An important positive observation from the case studies is that the involved designers all reported a better understanding of the systems. Ultimately, the DSE models are envisioned to play an important role in documenting a design.

7.10 Discussion and Conclusions

This chapter has presented the Octopus view on model-driven design-space exploration (DSE) for software-intensive systems, elaborating on the DSE process and envisioned tool support for this process. Model-driven DSE supports the systematic evaluation of design choices early in the development. It has the potential to replace or complement the spreadsheet-type analysis typically done nowadays.

Model-driven DSE can thus reduce the number of design iterations, improve product quality, and reduce cost.

To facilitate the practical use of model-driven DSE, we believe it is important to leverage the possibilities and combined strengths of the many existing languages and tools developed for modelling and analysis, and to present them to designers through domain-specific abstractions. We therefore set out to develop the Octopus DSE framework that intends to integrate languages and tools in a unifying framework. DSEIR, an intermediate representation for DSE, plays a central role in connecting tools and techniques in the DSE process in a flexible, extensible way, encouraging reuse of tools and of models. DSEIR allows to integrate domain-specific modelling, different analysis and exploration techniques, and diagnostics tools in customisable tool chains. Through DSEIR, model consistency and a consistent interpretation and representation of analysis results can be safeguarded. The current prototype tools combine simulation and model checking in the Octopus framework. DPML, a domain-specific modelling language for printer data paths, has been developed to provide support for the professional printing domain. The first industrial experiences with the Octopus approach in the printing domain have been successful.

Several challenges and directions for future work remain, both scientific challenges and challenges related to industrial adoption of model-driven DSE.

First of all, DSEIR needs further validation, also in other domains. It is already clear that extensions are needed, so that it covers all aspects of the DSE process. In particular, besides design alternatives and experiments, it would be beneficial to standardise the language for phrasing DSE questions and the format capturing DSE results. This would further facilitate the exchange of information between tools and the consistent interpretation of results.

Another direction for future research are the model transformations to and from DSEIR. The three transformations to analysis tools presented in this chapter are all of a different nature. It is important to precisely define the types of analysis supported by a transformation, the properties preserved by the transformation, and its limitations. Also techniques to facilitate (semi-)automatic translations and maintenance of transformations are important, to cope with changes in the Octopus framework or the targeted analysis tools.

Given precisely defined modelling languages and model transformations, it becomes interesting to explore integration of analysis techniques. Can we effectively combine the strengths of different types of analysis, involving for example model checking, simulation, and dataflow analysis? What about adding optimisation techniques such as constraint programming and SAT/SMT solving? No single analysis technique is suitable for all purposes. Integration needs to be achieved without resorting to one big unified model, because such a unified model will not be practically manageable. But how do we provide model consistency when combining multiple models and analysis techniques?

On a more fundamental level, integration of techniques leads to the question how to properly handle combinations of discrete, continuous, and probabilistic aspects. Such combinations materialise from combinations of timing aspects, user interac-

tions, discrete objects being manipulated, physical processes being controlled, failures, wireless communication, etc.

Scalability of analysis is another important aspect. Many of the analysis techniques do not scale to industrial problems. Is it possible to improve scalability of individual techniques? Can we support modular analysis and compositional reasoning across analysis techniques, across abstraction levels, and for combinations of discrete, continuous, and probabilistic aspects?

Early in the design process, precise information on the workloads of tasks to be performed and on the platform components to be used is often unavailable. Environment parameters and user interactions may further be uncontrollable and unpredictable. How can we cope with uncertain and incomplete information? How do we guarantee robustness of the end result of DSE against (small) variations in parameter values? Can we develop appropriate sensitivity analysis techniques?

The increasingly dynamic nature of modern embedded systems also needs to be taken into account. Today's systems are open, connected, and adaptive in order to enrich their functionality, enlarge their working range and extend their life time, to reduce cost, and to improve quality under uncertain and changing circumstances. System-level control loops play an increasingly important role. What is the best way to co-design control and embedded hardware and software?

To achieve industrial acceptance, we need systematic, semi-automatic DSE methods that can cope with the complexity of next generations of high-tech systems. These methods should be able to cope with the many different use cases that a typical embedded platform needs to support, and the trade-offs that need to be made between the many objectives that play a role in DSE. Model versioning and tracking of decision making need to be supported. Model calibration and model validation are other important aspects to take into account.

Model-driven DSE as presented in this chapter aims to support decision making early in the development process. It needs to be connected to other phases in development such as coding and code generation, hardware synthesis, and possibly model-based testing. Industrially mature DSE tools are a prerequisite. DSL support, tool chain customisation, integration with other development tools, and training all need to be taken care of.

In conclusion, the views and results presented in this chapter provide a solid basis for model-driven DSE. The motivation for the work comes from important industrial challenges, which in turn generate interesting scientific challenges. It is this combination of industrial and scientific challenges that makes the work particularly interesting. The scientific challenges point out the need for integration of modelling and analysis techniques, and not necessarily the further development of specialised techniques. A stronger focus on integration would benefit the transfer of academic results to industrial practice.

Acknowledgements This work has been 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, Agriculture, and Innovation under the BSIK program.

References

1. AADL (2012). URL http://www.aadl.info/. Accessed October 2012
2. Alur, R., Courcoubetis, C., Dill, D.L.: Model-checking in dense real-time. Inf. Comput. **104**, 2–34 (1993)
3. Alur, R., Courcoubetis, C., Halbwachs, N., Henzinger, T.A., Ho, P.H., Nicollin, X., Olivero, A., Sifakis, J., Yovine, S.: The algorithmic analysis of hybrid systems. Theor. Comput. Sci. **138**, 3–34 (1995)
4. Alur, R., Dill, D.L.: A theory of timed automata. Theor. Comput. Sci. **126**, 183–235 (1994)
5. Apache Felix (2012). URL http://felix.apache.org/. Accessed October 2012
6. Balarin, F., Giusto, P., Jurecska, A., Passerone, C., Sentovich, E., Tabbara, B., Chiodo, M., Hsieh, H., Lavagno, L., Sangiovanni-Vincentelli, A., Suzuki, K.: Hardware-Software Co-design of Embedded Systems: The POLIS Approach. Kluwer Academic Publishers, Norwell, MA (1997)
7. Basten, T., Hendriks, M., Somers, L., Trčka, N.: Model-driven design-space exploration for software-intensive embedded systems (extended abstract). In: M. Jurdzinski, D. Nickovic (eds.) Formal Modeling and Analysis of Timed Systems, *Lecture Notes in Computer Science*, vol. 7595, pp. 1–6. Springer, Berlin (2012)
8. Basten, T., van Benthum, E., Geilen, M., Hendriks, M., Houben, F., Igna, G., Reckers, F., de Smet, S., Somers, L., Teeselink, E., Trčka, N., Vaandrager, F., Verriet, J., Voorhoeve, M., Yang, Y.: Model-driven design-space exploration for embedded systems: The Octopus toolset. In: T. Margaria, B. Steffen (eds.) Leveraging Applications of Formal Methods, Verification, and Validation, *Lecture Notes in Computer Science*, vol. 6415, pp. 90–105. Springer, Heidelberg (2010). URL http://dse.esi.nl/. Accessed October 2012
9. Basu, A., Bozga, M., Sifakis, J.: Modeling heterogeneous real-time components in BIP. In: Proceedings of the Fourth IEEE International Conference on Software Engineering and Formal Methods (SEFM 2006), pp. 3–12 (2006)
10. Behrmann, G., David, A., Larsen, K.G., Hakansson, J., Pettersson, P., Yi, W., Hendriks, M.: Uppaal 4.0. In: Proceedings of the Third International Conference on the Quantitative Evaluation of Systems (QEST06), pp. 125–126 (2006). URL http://www.uppaal.com/. Accessed October 2012
11. Bleuler, S., Laumanns, M., Thiele, L., Zitzler, E.: PISA – a platform and programming language independent interface for search algorithms. In: C.M. Fonseca, P.J. Fleming, E. Zitzler, K. Deb, L. Thiele (eds.) Evolutionary Multi-Criterion Optimization, *Lecture Notes in Computer Science*, vol. 2632, pp. 494–508. Springer, Berlin (2003). URL http://www.tik.ee.ethz.ch/pisa/. Accessed October 2012
12. Börner, K.: Plug-and-play macroscopes. Commun. ACM **54**, 60–69 (2011)
13. Bulychev, P.E., David, A., Larsen, K.G., Mikučionis, M., Poulsen, D.B., Legay, A., Wang, Z.: UPPAAL-SMC: Statistical model checking for priced timed automata. In: Proceedings of the 10th Workshop on Quantitative Aspects of Programming Languages and Systems (QAPL 2012), pp. 1–16 (2012)
14. Cassez, F., Larsen, K.: The impressive power of stopwatches. In: C. Palamidessi (ed.) CONCUR 2000 Concurrency Theory, *Lecture Notes in Computer Science*, vol. 1877, pp. 138–152. Springer, Berlin (2000)
15. CoFluent Design, CoFluent Studio (2012). URL http://www.cofluentdesign.com/. Accessed October 2012
16. Cyberinfrastructure shell (2012). URL http://cishell.org/home.html. Accessed October 2012
17. Davare, A., Densmore, D., Meyerowitz, T., Pinto, A., Sangiovanni-Vincentelli, A., Yang, G., Zeng, H., Zhu, Q.: A next-generation design framework for platform-based design. In: Proceedings of the 2007 Design and Verification Conference (DVCon 2007) (2007)
18. David, A., Larsen, K.G., Legay, A., Mikučionis, M., Poulsen, D.B., van Vliet, J., Wang, Z.: Statistical model checking for networks of priced timed automata. In: Formal Modeling and Analysis of Timed Systems, *Lecture Notes in Computer Science*, vol. 6919, pp. 80–96. Springer, Berlin (2011)

19. Derler, P., Lee, E.A., Matic, S.: Simulation and implementation of the PTIDES programming model. In: Proceedings of the 12th IEEE/ACM International Symposium on Distributed Simulation and Real-Time Applications (DS-RT '08), pp. 330–333 (2008)
20. Eker, J., Janneck, J.W.: CAL language report specification of the CAL actor language. ERL Technical Memo UCB/ERL M03/48, University of California, Berkeley, CA (2003)
21. Eker, J., Janneck, J.W., Lee, E.A., Liu, J., Liu, X., Ludvig, J., Neuendorffer, S., Sachs, S., Xiong, Y.: Taming heterogeneity - the Ptolemy approach. Proc. IEEE **91**, 127–144 (2003)
22. Esterel Technologies, Scade Suite (2012). URL `http://www.esterel-technologies.com/products/scade-suite`. Accessed October 2012
23. Hendriks, M., Geilen, M., Basten, T.: Pareto analysis with uncertainty. In: Proceedings of the 9th IEEE/IFIP International Conference on Embedded and Ubiquitous Computing (EUC 2011), pp. 189–196 (2011)
24. Hendriks, M., Vaandrager, F.W.: Reconstructing critical paths from execution traces. In: Proceedings of the 10th IEEE/IFIP International Conference on Embedded and Ubiquitous Computing (EUC 2012) (2012)
25. Holzmann, G.J.: The SPIN Model Checker: Primer and Reference Manual. Addison-Wesley, Boston, MA (2004). URL `http://spinroot.com/`. Accessed October 2012
26. Houben, F., Igna, G., Vaandrager, F.: Modeling task systems using parameterized partial orders. Int. J. Softw. Tools Technol. Transf. (2012). Accepted for publication
27. Hsu, C.J., Keceli, F., Ko, M.Y., Shahparnia, S., Bhattacharyya, S.S.: DIF: An interchange format for dataflow-based design tools. In: A.D. Pimentel, S. Vassiliadis (eds.) Computer Systems: Architectures, Modeling, and Simulation, *Lecture Notes in Computer Science*, vol. 3133, pp. 3–32. Springer, Berlin (2004)
28. IBM ILOG CPLEX Optimizer (2012). URL `http://www.ibm.com/CPLEX/`. Accessed October 2012
29. Igna, G., Kannan, V., Yang, Y., Basten, T., Geilen, M., Vaandrager, F., Voorhoeve, M., de Smet, S., Somers, L.: Formal modeling and scheduling of data paths of digital document printers. In: F. Cassez, C. Jard (eds.) Formal Modeling and Analysis of Timed Systems, *Lecture Notes in Computer Science*, vol. 5215, pp. 170–187. Springer, Heidelberg (2008)
30. Igna, G., Vaandrager, F.: Verification of printer datapaths using timed automata. In: T. Margaria, B. Steffen (eds.) Leveraging Applications of Formal Methods, Verification, and Validation, *Lecture Notes in Computer Science*, vol. 6416, pp. 412–423. Springer, Heidelberg (2010)
31. Improvise (2012). URL `http://www.cs.ou.edu/~weaver/improvise/index.html`. Accessed October 2012
32. Jackson, D.: Software Abstractions: Logic, language, and Analysis. MIT Press, Cambridge, MA (2006)
33. Jackson, E.K., Kang, E., Dahlweid, M., Seifert, D., Santen, T.: Components, platforms and possibilities: towards generic automation for MDA. In: Proceedings of the Tenth ACM International Conference on Embedded Software (EMSOFT 2010), pp. 39–48 (2010)
34. Jensen, K., Kristensen, L., Wells, L.: Coloured Petri Nets and CPN Tools for modelling and validation of concurrent systems. Int. J. Softw. Tools Technol. Transf. **9**, 213–254 (2007)
35. Jensen, K., Kristensen, L.M.: Coloured Petri Nets: Modelling and Validation of Concurrent Systems. Springer, Berlin (2009)
36. JAVA Genetic Algoritms Package (2012). URL `http://jgap.sourceforge.net/`. Accessed October 2012
37. Katoen, J.P., Zapreev, I.S., Hahn, E.M., Hermanns, H., Jansen, D.N.: The ins and outs of the probabilistic model checker MRMC. In: Proceedings of the Sixth International Conference on the Quantitative Evaluation of Systems (QEST 2009), pp. 167–176 (2009)
38. Keinert, J., Streubühr, M., Schlichter, T., Falk, J., Gladigau, J., Haubelt, C., Teich, J., Meredith, M.: SystemCoDesigner–An automatic ESL synthesis approach by design space exploration and behavioral synthesis for streaming applications. ACM Trans. Des. Autom. Electron. Syst. **14**, Article No. 1 (2009). URL `http://www12.informatik.uni-erlangen.de/research/scd/`. Accessed October 2012

39. Kienhuis, B., Deprettere, E., Vissers, K., van der Wolf, P.: An approach for quantitative analysis of application-specific dataflow architectures. In: Proceedings of the 1997 IEEE International Conference on Application-Specific Systems, Architectures and Processors (ASAP '97), pp. 338–349 (1997)
40. Kumar, A.: Adding schedulability analysis to the Octopus toolset. Master's thesis, Eindhoven University of Technology, Faculty of Mathematics and Computer Science, Design and Analysis of Systems group, Eindhoven (2011)
41. Lee, E.A., Messerschmitt, D.G.: Static scheduling of synchronous data flow programs for digital signal processing. IEEE Trans. Comput. **36**, 24–35 (1987)
42. Lukasiewycz, M., Glaß, M., Reimann, F., Teich, J.: Opt4j: Meta-heuristic optimization framework for Java. In: Proceedings of the 13th Annual Conference Genetic and Evolutionary Computing Conference (GECCO 2011), pp. 1723–1730 (2011). URL `http://opt4j.sourceforge.net/`. Accessed October 2012
43. MathWorks - Global Optimization Toolbox - Solve multiple maxima, multiple minima, and nonsmooth optimization problems (2012). URL `http://www.mathworks.com/products/global-optimization`. Accessed October 2012
44. MathWorks - SimEvents - Discrete-Event Simulation Software (2012). URL `http://www.mathworks.com/products/simevents/`. Accessed October 2012
45. MathWorks - Simulink - Simulation and Model-Based Design (2012). URL `http://www.mathworks.com/products/simulink/`. Accessed October 2012
46. MLDesign Technologies, MLDesigner (2012). URL `http://www.mldesigner.com/`. Accessed October 2012
47. Modelica and the Modelica Association (2012). URL `http://www.modelica.org/`. Accessed October 2012
48. Moily, A.: Supporting design-space exploration with synchronous data flow graphs in the Octopus toolset. Master's thesis, Eindhoven University of Technology, Faculty of Mathematics and Computer Science, Software Engineering and Technology group, Eindhoven (2011)
49. Nikolov, H., Thompson, M., Stefanov, T., Pimentel, A., Polstra, S., Bose, R., Zissulescu, C., Deprettere, E.: Daedalus: Toward composable multimedia MP-SoC design. In: Proceedings of the 45th annual Design Automation Conference (DAC 2008), pp. 574–579 (2008). URL `http://daedalus.liacs.nl/`. Accessed October 2012
50. NuSMV (2012). URL `http://nusmv.fbk.eu/`. Accessed October 2012
51. Open SystemC Initiative (OSCI) (2012). URL `http://www.systemc.org/`
52. OSGi Alliance: Osgi service platform release 4 (2012). URL `http://www.osgi.org/Specifications/HomePage`. Accessed October 2012
53. Pareto, V.: Manual of political economy (manuale di economia politica). Kelley, New York (1971 (1906)). Translated by A. S. Schwier and A. N. Page
54. PRISM (2012). URL `http://www.prismmodelchecker.org/`. Accessed October 2012
55. ProM - Process mining workbench (2012). URL `http://www.promtools.org/prom6/`. Accessed October 2012
56. Qt - A cross-platform application and UI framework (2012). URL `http://qt.nokia.com/products/`. Accessed October 2012
57. RTCtoolbox: Modular Performance Analysis with Real-Time Calculus (2012). URL `http://www.mpa.ethz.ch/Rtctoolbox/`. Accessed October 2012
58. Sander, I., Jantsch, A.: System modeling and transformational design refinement in ForSyDe. IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst. **23**, 17–32 (2004)
59. Schindler, K.: Measurement data visualization and performance visualization. Internship report, Eindhoven University of Technology, Department of Mathematics and Computer Science (2008)
60. Stuijk, S., Geilen, M., Basten, T.: Throughput-buffering trade-off exploration for cyclo-static and synchronous dataflow graphs. IEEE Trans. Comput. **57**, 1331–1345 (2008)
61. Stuijk, S., Geilen, M., Theelen, B., Basten, T.: Scenario-aware dataflow: Modeling, analysis and implementation of dynamic applications. In: Proceedings of the 2011 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS XI), pp. 404–411 (2011)

62. Stuijk, S., Geilen, M.C.W., Basten, T.: SDF^3: SDF For Free. In: Proceedings of the 6th International Conference on Application of Concurrency to System Design (ACSD 2006), pp. 276–278 (2006)
63. Symtavision SymTA/S (2012). URL http://www.symtavision.com/symtas.html/. Accessed October 2012
64. SysML (2012). URL http://www.sysml.org/. Accessed October 2012
65. Teeselink, E., Somers, L., Basten, T., Trčka, N., Hendriks, M.: A visual language for modeling and analyzing printer data path architectures. In: Proceedings of the Industry Track of Software Language Engineering 2011 (ITSLE 2011), pp. 1–20 (2011)
66. Theelen, B.D., Florescu, O., Geilen, M.C.W., Huang, J., van der Putten, P.H.A., Voeten, J.P.M.: Software/hardware engineering with the parallel object-oriented specification language. In: Proceedings of the 5th IEEE/ACM International Conference on Formal Methods and Models for Codesign (MEMOCODE '07), pp. 139–148 (2007)
67. Theelen, B.D., Geilen, M.C.W., Basten, T., Voeten, J.P.M., Gheorghita, S.V., Stuijk, S.: A scenario-aware data flow model for combined long-run average and worst-case performance analysis. In: Proceedings of the Fourth ACM and IEEE International Conference on Formal Methods and Models for CoDesign (MEMOCODE 2006), pp. 185–194 (2006)
68. TimeDoctor (2012). URL http://sourceforge.net/projects/timedoctor/. Accessed October 2012
69. Trčka, N., Hendriks, M., Basten, T., Geilen, M., Somers, L.: Integrated model-driven design-space exploration for embedded systems. In: Proceedings of the International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS XI), pp. 339–346 (2011). URL http://dse.esi.nl/. Accessed October 2012
70. Trčka, N., Voorhoeve, M., Basten, T.: Parameterized timed partial orders with resources: Formal definition and semantics. ES Report ESR-2010-01, Eindhoven University of Technology, Department of Electrical Engineering, Electronic Systems group, Eindhoven (2010)
71. Trčka, N., Voorhoeve, M., Basten, T.: Parameterized partial orders for modeling embedded system use cases: Formal definition and translation to coloured Petri nets. In: Proceedings of the 11th International Conference on Application of Concurrency to System Design (ACSD 2011), pp. 13–18 (2011)
72. UML - Object Management Group (2012). URL http://www.uml.org. Accessed October 2012
73. UML Profile for MARTE: Modeling and Analysis of Real-Time and Embedded Systems (2012). URL http://www.omgmarte.org/. Accessed October 2012
74. van Beek, D.A., Collins, P., Nadales, D.E., Rooda, J.E., Schiffelers, R.R.H.: New concepts in the abstract format of the compositional interchange format. In: Proceedings of the 3rd IFAC Conference on Analysis and Design of Hybrid Systems (ADHS 2009), pp. 250–255 (2009)
75. Viskic, I., Yu, L., Gajski, D.: Design exploration and automatic generation of MPSoC platform TLMs from Kahn Process Network applications. In: Proceedings of the ACM SIGPLAN/SIGBED 2010 Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES '10), pp. 77–84 (2010)
76. Yang, Y.: Exploring resource/performance trade-offs for streaming applications on embedded multiprocessors. Ph.D. thesis, Eindhoven University of Technology, Eindhoven (2012)
77. Yang, Y., Geilen, M., Basten, T., Stuijk, S., Corporaal, H.: Exploring trade-offs between performance and resource requirements for synchronous dataflow graphs. In: Proceedings of the 7th IEEE/ACM/IFIP Workshop on Embedded Systems for Real-Time Multimedia (ESTIMedia 2009), pp. 96–105 (2009)
78. Yang, Y., Geilen, M., Basten, T., Stuijk, S., Corporaal, H.: Automated bottleneck-driven design-space exploration of media processing systems. In: Proceedings of the Conference on Design, Automation and Test in Europe (DATE 2010), pp. 1041–1046 (2010)
79. Yang, Y., Geilen, M., Basten, T., Stuijk, S., Corporaal, H.: Playing games with scenario- and resource-aware SDF graphs through policy iteration. In: Proceedings of the Conference on Design, Automation and Test in Europe (DATE 2012), pp. 194–199 (2012)
80. Yices: An SMT Solver (2012). URL http://yices.csl.sri.com/. Accessed October 2012