A Visual Language for Modeling and Analyzing Printer Data Path Architectures

Egbert Teeselink[1], Lou Somers[1,2], Twan Basten[1,3], Nikola Trčka[1], Martijn Hendriks[4]

[1]Eindhoven University of Technology, [2]Océ Technologies, [3]Embedded Systems Institute, [4]Radboud University Nijmegen

contact: lou.somers@oce.com (Lou Somers)

Abstract

The data path is an important printer component that performs real-time image processing. Because of the large data sizes and high throughput requirements in high-performance multifunctional printers, the data path is usually implemented as a hybrid software/hardware system. Designing the architecture of a data path is a nontrivial problem, because of the many tradeoffs involved and because it is difficult to analyze how well a design conforms to many of the important quality attributes. One quality attribute that is difficult to deduce from a data path design by hand is its throughput, typically expressed as the amount of pages that it can handle per minute. Because this is hard to analyze by hand, it is also difficult to predict how well a data path will perform when requirements change. This limits how structured and flexible the data path design process can be.

In order to improve this process, a solution is needed that helps architects to analyze the behavior of a model of a data path architecture, find its throughput and identify bottlenecks. The solution must be sustainable, so that support for increasingly detailed architecture models can be added.

Our aim is to enable these goals by providing a domain-specific modeling language for data path architectures, a modeling environment that supports this language, and a set of tools for model conversion and analysis. The language also allows transformations to timed automata, colored Petri nets and synchronous data flow graphs. This paper concentrates on the modeling language and its design rationale.

1. Domain analysis

The *data path* of a printer consists of the image processing steps that need to take place on the stages between for example network input and the print head (network printing) or between the scanner and the print head (copying). Figure 1–1 shows the place of the data path in a heavily simplified illustration of a multifunctional printer.

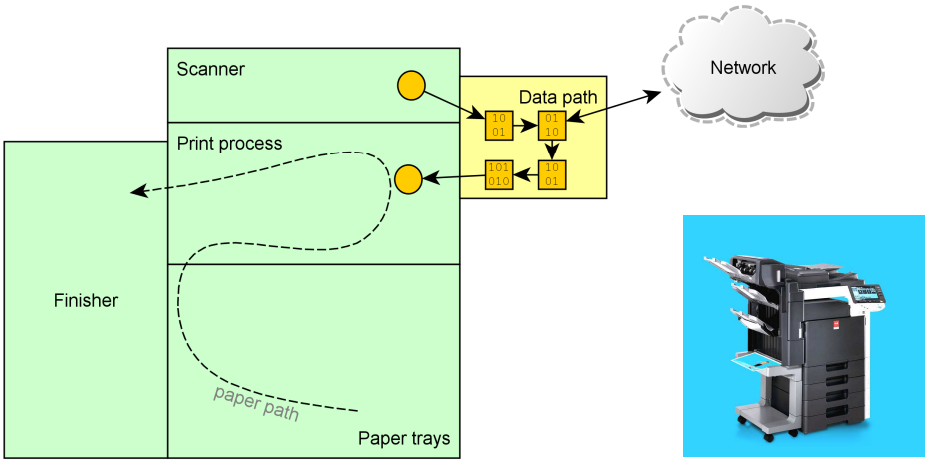

Figure 1–1. A schematic illustration of a multifunctional printer, showing the place of the data path with respect to the print process, the paper path, the scanner and the network. To the right, a typical multifunctional office printer with a similar layout is shown.

The data path performs necessary image processing steps to improve quality, to conform to the user's settings but also to overcome limitations of scanning and printing hardware [3]. For example, when scanning a white page with text, the user expects an image that has a white background. Scanners, however, typically detect a light shade of gray, rather than white. Figure 1–2 shows how image processing can help to produce the desired results.

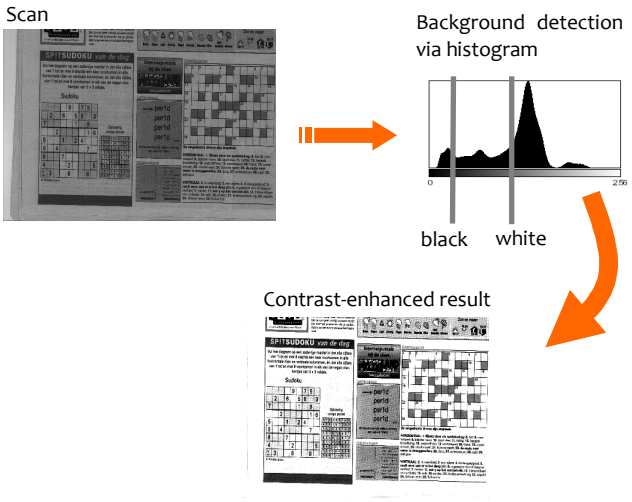

Figure 1–2. An example of how background detection and contrast enhancement improve the quality of a scan; adapted from [3].

Depending on the use case and its parameters, a variety of image processing steps have to be performed, which can be supported in all kinds of ways. For example, the data path for copying on an imaginary printer could be structured as shown below [3].

Figure 1–3. The steps that a data path may need to perform for a copy use case.

Image processing steps such as the ones described above can be implemented in a variety of ways. For example, they can be executed on an FPGA, on a generic CPU, or on a DSP. When choosing a processing chip for such image processing steps, each option has advantages and disadvantages in terms of, for instance, cost, speed, latency, and power usage. Similar trade-offs apply to the whole configuration of processing units and the attached memories and data channels.

Closely related to the hardware choice is the order in which the various image processing steps are performed, and exactly how these steps are expected to behave. For example, if a scanned image needs to be both rotated and resized, these steps can be done in either order. However, if the image is usually scaled up and not down, the process may run faster if the rotation is performed first, on the smaller image. On the other hand, the rotation procedure may produce ugly artifacts, which may be less visible when performed on a higher-resolution image, so after the resize step. Such choices, and many more complicated ones, are typically closely tied to the hardware choices.

Designing a good data path, therefore, is a challenging task. Currently, architects often attempt to compute the performance of several alternatives manually and must subsequently create test set-ups for the alternatives that seem most promising, to validate their computations. This is a labor-intensive process, which impacts the time-to-market of new printer products.

Data path design choices are influenced by a number of criteria. Among these, one important criterion is the throughput of a design, typically expressed in terms of “the maximum number of pages per minute that the data path can handle”. In order to compare the throughput of different data path designs, a good estimate is required of the throughput of each design.

One way to estimate the throughput of a data path design, is by means of some manual calculations. An architect could compute the speed of the processors that perform the heaviest image processing task and get an indication of processing times. Similarly, he or she could compute the amount of data communicated between components per page and compare that to the bus bandwidths between those components, for an indication of the number of pages per minute that these buses are able to sustain. Putting these kinds of computations together, an estimate can be obtained of the maximum throughput of a data path design.

However, it turns out that these numbers do not correspond well to the real behavior of a data path, once it is built. This is because in the data path, there is a lot of dynamic behavior going on with emerging characteristics. These are difficult to predict with simple calculations.

For example, imagine two image processing steps, A and B, both of which transfer their output to a memory block over the same bus. Step A, when running at maximum speed, uses about 60% of the capacity of that bus. Step B uses about 70%. Additionally, step A has a higher priority, which means that if both A or B need a resource, A gets what it needs and then B gets whatever is left.

Now, imagine the scenario where first only B is running, but before B is complete, A starts. Such a scenario is shown in Figure 1–4.

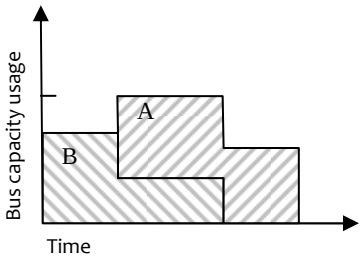

Figure 1–4. An example of dynamic resource sharing

In this case, the bus capacity for step B is limited, once step A kicks in. This means that B will complete later than what would have been expected when looked at in isolation. Additionally, any steps that can only start when B has completed, also get delayed, effectively delaying the entire processing of that particular page, thus slowing down the entire data path. To predict this kind of behavior, more powerful analysis tools are needed.

2. The tool chain

The problems described above can be solved by a means to analyze data path architectures in more detail. Such a solution has two main purposes: to find the *maximum throughput* of a data path design and to find *bottlenecks* in the design. Bottlenecks are those parts of the data path that, when removed, would allow the throughput to be higher. These are the most interesting parts, because changing them yields the largest effect in terms of throughput. Nevertheless, in order to understand why the bottlenecks are where they are, it is usually useful to obtain a more thorough understanding of the entire data path behavior as well.

It is also important that our solution is *sustainable* enough that it can be continuously improved for many years to come. This means that it must be maintainable and extensible, most notably in such a way that support can be added for better and more detailed models. Additionally, the solution must share concepts and technology with other tools, so that the results are easily integrated.

As described above, the difficulty in analyzing data path performance lies in its difficult-to-predict dynamic behavior. Therefore, we somehow need to reproduce that dynamic behavior and measure it. One way to do so is by making a model of the various components that constitute a data path, simulating that model, and measuring its behavior.

For a software program to be able to analyze data path designs, we must model these data paths in a way that the program can understand. Therefore, we need some modeling language as well as an editing environment in which these models can be made. Additionally, if architects are to effectively use this modeling language, it must be user-friendly and easy to learn. Finally, a way to visualize the analysis results is needed, so that the architects can interpret the analysis outcomes.

Many applications exist that allow one to model and measure the behavior of embedded hardware and software systems [4,5,6,7,10,11,13,14,17,18]. However, they are not suited particularly well for our needs. First of all, a custom solution can be made as simple as reasonably possible. A more generic solution typically includes a larger amount of concepts and generalizations. The very presence of such concepts and generalizations gives that solution the ability to deal with a large set of problem domains, but many of them may not be particularly useful for analyzing data path designs. Thus, such a solution has a steep learning curve, compared to a solution tailored to printer data paths. In other words, we would like our solution to be as domain-specific as possible.

Second, depending on a third party solution creates a dependency on the service level of its vendor. Whereas with large, reliable vendors, this is often an advantage rather than a disadvantage, many of the solutions found are entirely academic in nature or supported by small startup companies. A lack of vendor support could render the entire application unusable, so this is a major risk. A custom-

developed solution can be maintained internally, removing such a dependency. This does increase the importance of the maintainability requirement, however, which is in turn easier to meet because the solution can be simple and domain-specific.

Therefore we have chosen to develop a domain-specific modeling language with hooks to connect to existing analysis tools. Our tool chain has a typical pipe-and-filter architecture. It consists of a series of tools, each of which consumes some input and produces some output, which in turn forms the input for the next tool.

The tool chain is based on a custom domain-specific language, called the Data Path Modeling Language (DPML). DPML models can be created and edited in a custom DPML editor. These models can then be compiled to an intermediate language called DPML Compact. DPML Compact models are significantly simpler in structure than DPML models, and therefore simpler to analyze and transform, yet any model that can be expressed in DPML can be expressed in DPML Compact. Finally, the tool chain includes a custom simulator which can execute DPML Compact models and log events into trace files. These trace files can be visualized using ResVis [15].

2.1. DPML and the DPML editor

The Data Path Modeling Language, or DPML, is a combined visual and textual language that can be used to specify models of printer data paths. It has been designed particularly to help to measure the speed of data paths, which means that a lot of information about data path behavior can be omitted. Additionally, it has been designed to allow for modular and expressive models, that are as simple as the problem domain allows. DPML is discussed in detail in Section 3.

DPML models can be edited in a custom editor that is based on the open source Qt library [12]. A single DPML model is contained in multiple small files, each of which describes a small reusable element. A textual DPML element is stored as a plain text file and visual DPML elements are stored in a custom XML format.

2.2. DPML Compact and the compacter

As described above, DPML models consist of many small files. Additionally, they have a lot of implicit behavior, all of which has been designed to make the life of data path designers easier. To facilitate the analysis and further conversion of DPML models, we must first use a tool called the *compacter* to convert them to a simpler data format, DPML Compact.

This allows for a separation of concerns: The compacter tackles the problem of interpreting the *structure* of a DPML model, i.e. parsing and analyzing DPML files, discovering all implicit behavior and dealing with complex structural features. This is then entirely decoupled from the problem of analyzing the *behavior* of DPML models, which can be done based on the much simpler DPML Compact models.

The separation between DPML and DPML Compact has a positive impact on the maintainability and extensibility of the tool chain. By separating structural interpretation from behavioral analysis, each tool has a single, well-defined purpose. If a single tool would perform both tasks, it would likely be more complicated and thus more difficult to maintain. Additionally, when additional analysis tools are to be connected, the structural features of DPML can be entirely disregarded and the much simpler DPML Compact can be used as an input instead.

Finally, additional language features can often be added to DPML without any change to the analysis tools if they do not imply fundamentally new kinds of behavior. In such cases, only the editor and the compacter need to be changed. As the amount of available analysis tools is expected to increase, this is a major advantage.

2.3. Jobs and the simulator

The simulator analyzes the behavior of a DPML Compact model by executing it, given a *job* description. Such a job description captures the settings that a fictional user entered, such as the amount of pages to print, scan or copy, the zoom factor or the size and orientation of each page. These inputs are sufficient to predict the behavior of the modeled data path if it would be given the task of processing the specified job.

There is no predefined set of page properties that must be present in a job description. Instead, this choice is left up to the user: any property specified in the job can be referred to from the model. This is

a powerful concept, because it allows any variable characteristic or environmental factor that may impact the behavior of the data path being designed to be specified as a parameter instead of a constant.

The simulator measures the time that each image processing step takes and the amount of resources that it uses. It assigns resource capacity to any simultaneously running steps as dictated by rules specified in the model, and it ensures that each step only runs as fast as its total resource assignment allows.

The simulator keeps track of interesting events, such as resource assignment changes and processing steps that start and complete. These events are stored as trace files in a format understood by ResVis, a simple visualization tool designed for visualizing data path traces.

2.4. Other analysis tools

The DPML Compact models can also be transformed to DSEIR, the Design Space Exploration Intermediate Representation [2]. DSEIR is not intended to be a modeling language for data path designers to use directly, but merely to be a representation that can be easily transformed to many formalisms, yet captures any data path design that architects would want to measure. DPML Compact lies closer to the data path problem domain, and DSEIR is intended to be more generic.

DSEIR models can be used as input for simulation and model checking tools like SDF3 (synchronous data flow graphs) [16], CPN Tools (colored Petri nets) and Uppaal (timed automata). CPN Tools [8] can perform simulations of Colored Petri Nets. Using stochastic analysis methods, it allows for the user to not fully specify each property of a job or the platform, and instead merely specify stochastic parameters for it. Thus, unknowns can be estimated with an error margin and with stochastic analysis, the user may still get an insight in the behavior of the system. Uppaal [1] is a model checker for timed automata. By not specifying some scheduling rules in a data path, the user effectively creates a non-deterministic model. For example, the user may choose to leave task priorities unspecified. Uppaal can then be used to find the most efficient execution of that model, choosing any prioritization it sees fit. From this, in turn, optimal priority rules may be derived.

2.5. A chain of independent tools

Our tool chain is not an integrated environment; each tool is an independent program that takes some input and produces some output. Currently, these have to be manually started from the command line. Not tightly integrating each separate tool was a conscious choice, however, because it provides for a number of advantages:

- Each tool is forced to have a single, unique purpose. This makes maintenance easier, because for many changes and fixes, only one or two tools need changes. Not having to learn and understand the structure of a single monolithic system makes making such changes simpler
- Interfaces at tool boundaries (the data formats exchanged) are well defined and language independent. This makes it easy to create additional tools, even in different programming languages. This means that architecturally, the tool chain is highly extensible.
- Communicating intermediate data through well defined, human-editable data formats and allowing the user to invoke each step manually allows for human modifications at any point while walking through the tool chain. This can be very useful in the case of partly supported features or malfunctioning tools.
- Finally, a side-effect of forcing a manually operated tool chain upon the user is that he or she is forced to understand what each tool does, at least to some extent. Given that the projected user base is small and that the same users are expected to be involved in maintenance, such understanding may be vital to any maintenance work being successfully performed at all.

Apart from these reasons, we observe that all projected users are skilled computer users, who are comfortable typing commands on a command line and creating rudimentary batch files to automate repetitive tasks.

3. The Data Path Modeling Language

Our domain-specific language to analyze the performance of printer data path designs is called DPML (Data Path Modeling Language). When designing DPML, four goals were kept in mind.

First, DPML must be particularly suited to analyzing 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 behavioral issues that do not influence speed may be omitted.

Furthermore, DPML has to be **expressive** and **flexible**. This means that it must be possible to express a wide variety of models, with different behavioral and structural properties. This is important, because we cannot always foresee what kinds of designs engineers may want to analyze in the future, or what other purposes (than analyzing 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 also to **closely match the problem domain**. This means that all elements commonly found in data paths must be well supported and easy to model. Without this, even simple designs would require considerable modeling effort and thus a steeper learning curve for people using the tool chain for the first time. This may in turn impact the adoption of the tool chain as a means to improve the data path design process.

Finally, DPML has to support **modular** designs. This way, parts or elements that are used in multiple designs can be reused, thus saving modeling 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.

3.1. The structure of DPML

The Data Path Modeling Language is a combined visual and textual Domain Specific Language (DSL). 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 little element in a data path design, such as the behavior 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 to the language.

Structurally, a complete DPML model has three distinct components:

- 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.

This separation of concerns is based on an approach called the “Y-Chart approach” [9], which is commonly used when analyzing embedded system designs with models. Figure 3–1 graphically depicts this approach. The idea is that by separating the platform from the application, it is easy for a modeler to substitute one application for another. This way, one can measure how well a certain platform performs for a whole array of applications. Similarly, one can measure the performance of an important application on a set of alternative platform designs and find out which platform performs best.

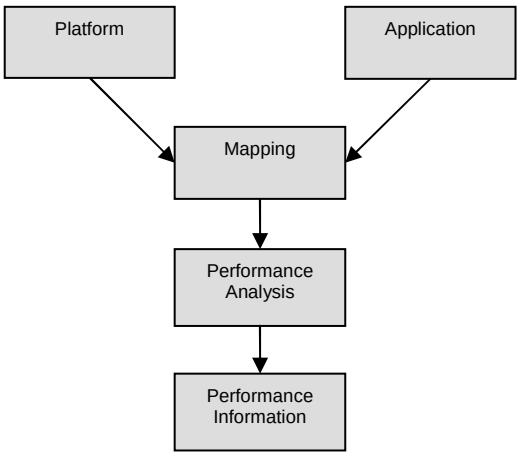

Figure 3–1. The Y-Chart

An engineer must still manually define a mapping for each combination of an application and a platform that he wants to analyze, however. This may be a tedious job if a lot of combinations must be tested. Therefore, it is important that the mapping contains as little information as necessary and that parts of it can be reused among other, comparable, mappings as much as possible.

3.2. The application

A printer data path typically has to be able to support a number of common use cases, such as “Copy a stack of pages three times”, “Print a document double-sided”, or “Scan a document to a PDF file and email it”. Each of these use cases requires the data path to perform a different series of steps, which is what we specify in the application. For example, a copy or scan use case may require additional contrast and color enhancement steps before the image can be safely printed or sent to the user.

3.2.1. A copy application in DPML

Figure 3–2 shows an example of a use case modeled in DPML.

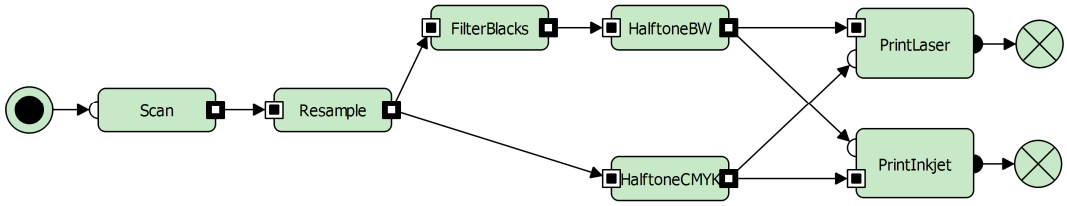

Figure 3–2. A DPML application called `corp.tonerjet.Copy`.

Every rounded rectangle in Figure 3–2 is a *step*, which is a single image processing operation that the data path must perform when executing this application. As you can see, each step has two or more *pins*, the squares or circles on the sides. The white pins are input pins and the black pins are output pins, and arcs can be drawn from output pins to input pins.

Pins are typed, and can have three possible data 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 (▣, ◘). We say that 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, then it means that this step *produces data* at that pin. If it is a `PixelData` pin, this additionally implies that the data produced is a bitmap image. Similarly, when a step has an input `Data` pin, it means that this *consumes data* at that pin. This way, we can specify that the `Scan` step in Figure 3–2 produces a bitmap image (through its `PixelData` output pin) which `Resample` consumes.

Arcs between steps have a second purpose, however: they 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 also explains the need for `Simple` input pins; no data is consumed or produced on those pins, but they can still be used to fix the execution order between steps. In Figure 3–2, this is

used to describe that `PrintInkjet` has to wait until `HalftoneBW` is done, even though `PrintInkjet` does not consume the image data produced by `HalftoneBW` (and similarly for `PrintLaser` and `HalftoneCMYK`).

3.2.2. Step behavior

The `corp.tonerjet.Copy` application in Figure 3–2 gives us a good idea about how exactly we want the copy use case to be implemented. However, for doing automated analysis of a data path design we need more information than just the execution order and the data flow between steps: we need to know the *behavior* of steps.

Recall that we are only interested in analyzing the *speed* of data path designs. Because of this, we do not necessarily need to specify all aspects of the behavior of steps, which would be a complicated ordeal indeed. Instead, we can do with just the information needed for analyzing the running time of a single step execution.

Because the data path is all about image processing, we can make an additional assumption: we assume that we can always model the duration of a single image processing step as a function of the size of an image, 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's behavior, we can do with just specifying the *image sizes* that it produces.

3.2.3. Tasks

In DPML, a *task* is used to describe a single operation that we might 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 belongs to exactly one task. It is, however, possible for multiple steps to belong to the same task. 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, respectively, classes and objects in many object-oriented programming languages.

Tasks are stored as small text files with content like in Listing 3–3.

```
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
}
```

Listing 3–3. An example: the `corp.tonerjet.Resample` task

3.2.3.1. Header and pin declarations

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 amount of input and output pins, their names and their data types. Notice how the pins are declared in the task, and not in the steps of the application itself: the amount of data objects consumed and produced is an inherent property of the behavior of a task itself, and not of the application as a whole. Therefore, the textual definition of a task influences the visual appearance of an associated step in an application. A 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`.

3.2.3.2. Task properties

The last part of a task is a set of *properties*. These constitute a functional description of its behavior. Because we assume that the actual content of the image produced does not matter (as described in Section 3.2.2), 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 behavior). Because the output pin in Listing 3–3 is of type `PixelData`, we specify the size of the produced image by specifying its dimensions and its bit depth.

In the example of a resample task, the amount by which an image is scaled depends on a use setting: effectively the resample step in a data path implements the "zoom" feature of a printer. Therefore, 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 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. The `page` keyword always refers to 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 (the `precondition` property in Listing 3–3). All of these other properties are optional.

3.2.3.3. Rationale

Tasks are separate from steps for two reasons. First of all, a common use case is where an engineer wants to compare multiple slightly different applications on the same hardware. If all behavior description would be contained inside the application, instead of separate entities such as tasks, then these different applications would contain a lot of duplicate code. This makes making changes in all applications cumbersome.

Second, even inside a single application it is possible for multiple steps to exhibit the same behavior. For example, in the `corp.tonerjet.Copy` example of Figure 3–2, there are two steps that perform printing, `PrintLaser` and `PrintInkjet`. Functionally, all a print step does is to consume an image (and somehow put it on paper, but that it mostly outside the scope of the data path), so therefore these two steps may very well have a single task, such as the one in Listing 3–4.

```
task Print
{
  inpin in:   PixelData;
  inpin sync: Simple;
  outpin out: Simple;

  //no properties: there is no Data output pin
}
```

Listing 3–4. The `corp.tonerjet.Print` task.

3.2.4. Task and application meta-model

As explained above, applications and tasks together describe the functional behavior of a data path. The figure below shows how tasks, applications and their components relate.

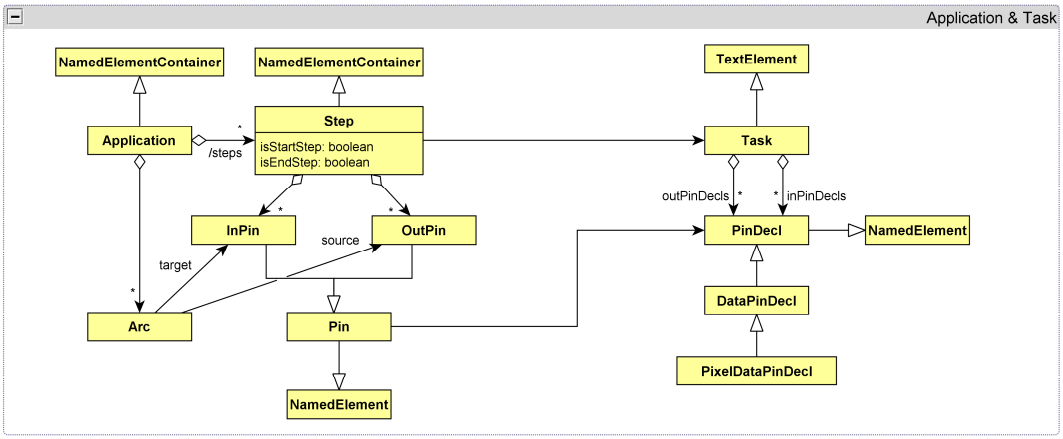

Figure 3–5. Task and application meta-model

3.3. The platform

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.

3.3.1. The structure of a platform model

In DPML, we summarize these components to a set of three resource types, as follows:

- 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, i.e. a maximum amount 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 semantically, there is no difference between e.g. a processor and a printer in DPML.

With just these blocks, we can create sufficiently realistic platform models for analyzing the performance of a data path design. As example, assume a possible platform that we might be considering: one based on an off-the-shelf PC, where most of the image processing happens on the CPU. The actual printing and scanning operations are real-time operations, however, which are implemented in a custom FPGA-based system-on-a-chip on a custom board and fit into the PC by means of a PCI Express bus. Figure 3–6 shows how such a platform could be modeled in DPML.

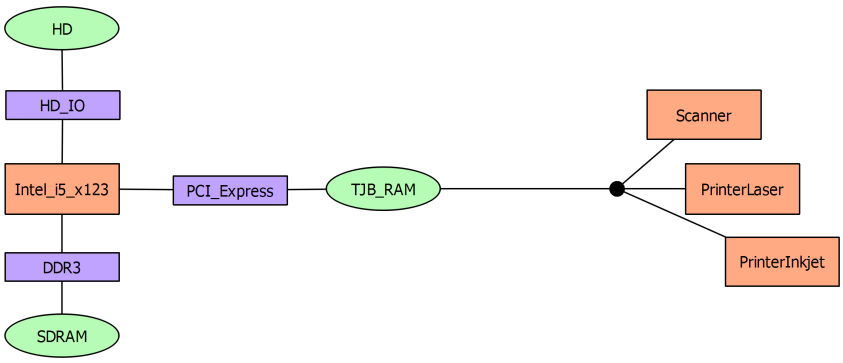

Figure 3–6. The `corp.tonerjet.PC` platform

In this platform, the column on the left is a simplified model of the standard PC architecture. The part on the right represents the custom board with a scanning resource and two printing resources. The nodes in a platform are called a *resource block*. As can be seen in Figure 3–6, memory blocks are shown as ellipses, bus blocks as thin rectangles, and executor blocks as larger rectangles.

Note that buses are the only components that can limit data transfer speed. Thus, the `HD` memory, which models a hard drive, can in fact read and write data infinitely fast. To model the fact that hard drives have limited I/O speeds, the `HD_IO` bus was added. This is not really a bus (or in fact, it models both the SATA/SCSI bus by means of which the hard drive is connected to the PC motherboard and the hard drive's inherent maximum read/write speed), but that does not matter: in DPML, anything that may limit data throughput is modeled with a bus.

3.3.1.1. Connections

The lines between the blocks in a platform model have semantics. Such a line is called a *connection*, and it means that the two (or more) blocks are directly connected to one another.

Connections limit the possible routes by which data can flow though the platform: for example, `Intel_i5_x123` can only read and write data in `SDRAM` via the bus block `DDR3`. An additional requirement is that on a route between an executor block and a memory block, there may only be bus blocks. This means that, for instance, in `corp.tonerjet.PC`, `PrinterInkjet` cannot directly read from `SDRAM`.

It is allowed for multiple routes between an executor and a memory to exist: if this is the case, a heuristic is used to choose which path is used.

3.3.2. Resources

Resource blocks have properties, much in the same way as the steps of an application have behavior. 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 that is on the market. A resource block based on that resource, then, 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.

3.3.2.1. Examples

Resources are typically simpler in structure than tasks: many resources only have one or two properties. Listing 3–7 lists how some of the resources used in the `corp.tonerjet.PC` platform of Figure 3–6 may be expressed.

```
processor CPU
{
  multitask = true;
}

processor Intel_i5_x123 isa CPU
{
  speed = 1G; //1 billion computations per second
}

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

memory SDRAM
{
  capacity = 2G; //2 billion bytes
}

scanner Scanner
{
  A4length = 210; //in millimeters
  pagesPerMinute = 100; //100 A4's per minute

  //page length that is scanned per second, in millimeters
  mmPerSecond = (pagesPerMinute * A4length) / 60;
}
```

Listing 3–7. Some resources for the TonerJet platform

3.3.2.2. Resource properties

As shown in Listing 3–7, each resource type has its own (small) set of properties. In most cases, just one property suffices. 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.

For `speed` property of processors, however, we cannot do this. Most processing units have a fairly standard way of measuring their speed, clock cycles, but exactly what can be done in a single clock cycle widely differs. An Intel CPU may be able to multiply two vectors in a clock cycle, whereas a modern GPU may be do the same with 128 vectors in a single clock tick. Moreover, for FPGAs the amount of work that can be done inside a single clock cycle as is much a design decision as a hardware property. In general, the meaning of the “speed” of a processor heavily depends on the operation that is being performed. Therefore, the `speed` property can have an arbitrary unit. In Section 3.4.2 we describe how the duration of each step can be computed using speeds of arbitrary units.

Finally, scanners and printers have no predefined properties at all: they only have *custom properties* that the user can choose to fill in. In Section 3.4.2 it is shown how these custom properties can be used to model printing and scanning behavior.

3.3.2.3. Inheritance

The `Intel_i5_x123` resource in Listing 3–7, has an additional property: it *is a* CPU. Resources in DPML can have parent resources, and a resource inherits all properties from its parent resource (but can override them). This is useful to avoid code duplication, but it can also severely reduce the amount of effort needed to write mappings. This is explained in detail in Section 3.4.2.1.

3.3.2.4. Rationale

The rationale for separating resources and resource blocks is roughly the same as the rationale for tasks and steps (as described in Section 3.2.3.3): to avoiding code duplication between and within platforms. However, for platforms a third important reason plays a role. In practice, teams sometimes discover that properties of hardware components differ from those advertised. If there is a chance that similar hardware components may be used in other projects running in parallel, wide usage of a shared repository of resource blocks could improve communication between projects about these kinds of discoveries.

Exactly how this repository may be used and shared is outlined in Section 3.5, but separate, independent elements that only describe the properties of a hardware component (separate from its role in a platform) are a prerequisite for such sharing to be effective.

3.3.3. Resource and platform meta-model

Platforms and resources are used together to describe the physical structure of a hardware platform that may be used to implement a data path. The figure below shows how tasks, applications and their components relate.

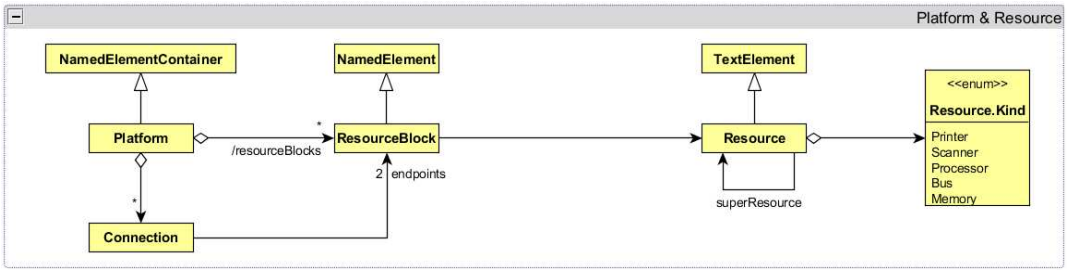

Figure 3–8. Resource and platform meta-model

3.3.4. Three resource types are enough

DPM platform models only have buses, memories and executors. This means there are no more specialized versions of such resources, such as hard drives or USB cables. Additionally, caches are not present in DPML either, not as properties of resource and neither as independent resource types. It turns out that currently, such elements are not needed, because of common properties shared by all known data path designs.

Recall that a data path is a component that performs image processing and transfer operations. Because of this observation, we can make a few additional assumptions:

1. Data transferred between steps is *new*, i.e. a step has not recently read or written exactly the same image.
2. Data is 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.

These assumptions together allow us to ignore the existence of caches entirely. We assume that the caches do not influence processing speed when an image is read from memory, because of assumptions 1 and 2: 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 that 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.

Finally, because the data is 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 (images), the *seek time* needed to move the read head to the right position is negligible.

When combining the assumptions above, we can conclude that with just buses, memories and executors, we can model a sufficiently complex computation platform for data paths.

Note that it is not impossible to design a data path in which one or more of these assumptions do not hold, and the analysis of such a DPML model may yield significantly different results than the real data path would. Therefore, it is important that users of DPML are aware of these assumptions. If it turns out that some of these assumptions are invalid more often than not, additional features may be added to DPML to overcome the issue. Nevertheless, communication with engineers involved in data path design showed that caches and detailed hard drive seek times are not typically the issues that limit the speed of data paths. Thus, in DPML, we ignore caches and detailed hardware behavior such as hard drive seek times.

Finally, note that due to the modular and extensible structure of DPML, it is relatively easy to overcome this assumption 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.

3.4. The mapping

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

3.4.1. Links

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.

3.4.1.1. Storage links

Recall that on each `Data` output pin of a step, the step produces new data. This data needs to be stored somewhere for a subsequent step to be able to read and process it. A storage link specifies where such data is stored, so it is a link between output `Data` pins and memory blocks. For our running example, map the `corp.tonerjet.Copy` application on the `corp.tonerjet.PC` platform. Let us assume that scanning and printing takes place on the custom *TJB* board. This means that the scanning and halftoning steps write data to the memory on the *TJB* board, `TJB_RAM`. The rest of the processing takes place on the PC, however, so all intermediate data is stored in the PC's `SDRAM`. Figure 3–9 shows how we can model this in DPML.

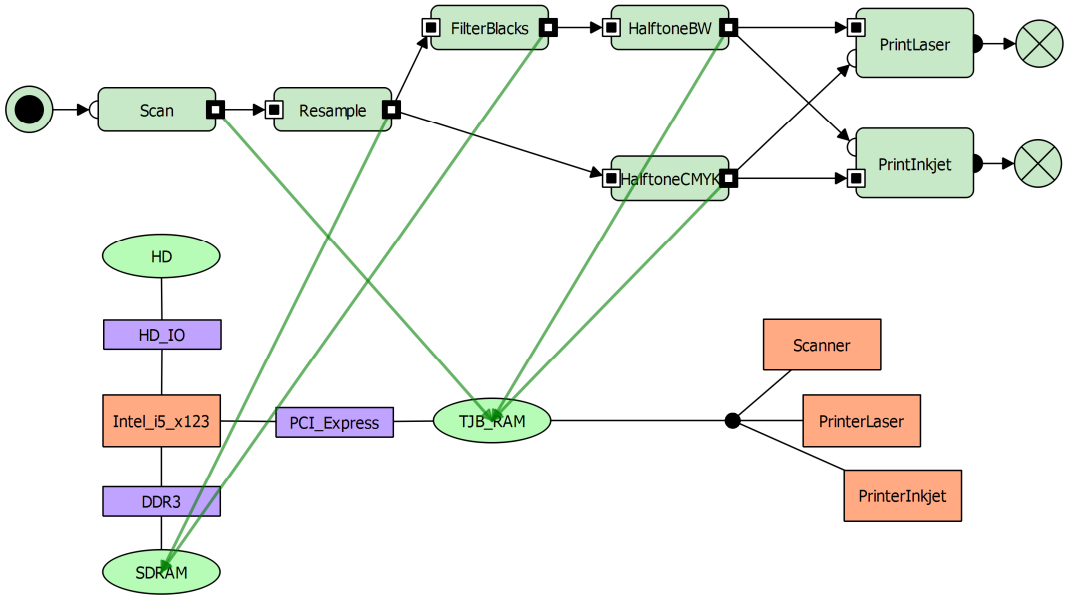

Figure 3–9. Storage links in the `corp.tonerjet.Copy_PC` mapping

3.4.1.2. Allocation links

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

There are two kinds of allocation links: claim links and release links. 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 on, and using the properties from the task associated to the output pin's step, it can be determined how much memory should be claimed.

For our example, let us assume that most steps claim their own memory, and that the following steps release that memory again. The only exception is the `Scan` step, we want to claim its own memory as well as the memory needed for resampling. This way, scanning a new page does not start if the main working memory of the data path is too full. Additionally, the memory of `Resample.out` could be released by two steps: `FilterBlacks` and `HalftoneCMYK`. We choose to make `HalftoneCMYK` responsible for the memory release, because it is expected to complete later. We can model all this in DPML like in Figure 3–10.

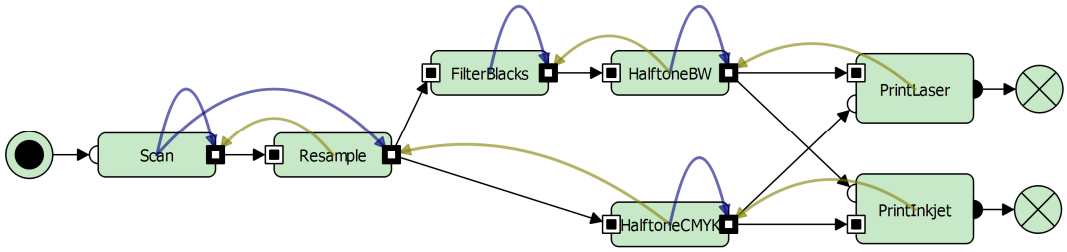

Figure 3–10. Allocation links in the `corp.tonerjet.Copy_PC` mapping

As you can see, this creates a very 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. With these rules in mind, we can simplify Figure 3–10 into the model shown in Figure 3–11.

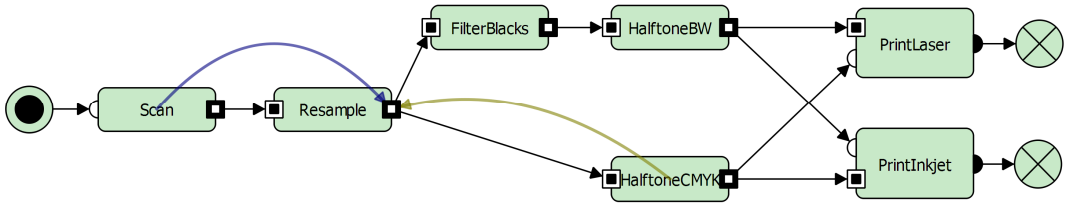

Figure 3–11. Only the necessary allocation links in the `corp.tonerjet.Copy_PC` mapping. This mapping has equivalent semantics to the one in Figure 3–10.

3.4.1.3. 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 is 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*, explained in Section 3.4.2, and a *priority*.

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. The exact meaning of priorities depends on the analysis tools used, however, as DPML is not designed for a single tool only. It is possible for multiple execution links to have the same priority, and this should imply that resources are somehow fairly shared between the two steps.

We do enforce one important convention with respect to priorities, however: the higher the number, the lower the priority. So a step with priority 3 is considered more important than a step with priority 7.

Recall that in our `corp.tonerjet.Copy_PC` mapping, we expect scanning and printing to be performed on the *TJB* board, and the other operations on the PC. Let us assume that we give scanning the lowest priority and printing the highest priority. This way, in case of resource scarcity, the data path is first “emptied” of pages that are currently being processed, before new pages are inserted into the processing pipeline. This avoids deadlocks or unnecessary delays. The rest of the steps are

prioritized according to the same rationale, so the later a step occurs in the pipeline, the higher its priority. A mapping such as this can be modeled in DPML like in Figure 3–12.

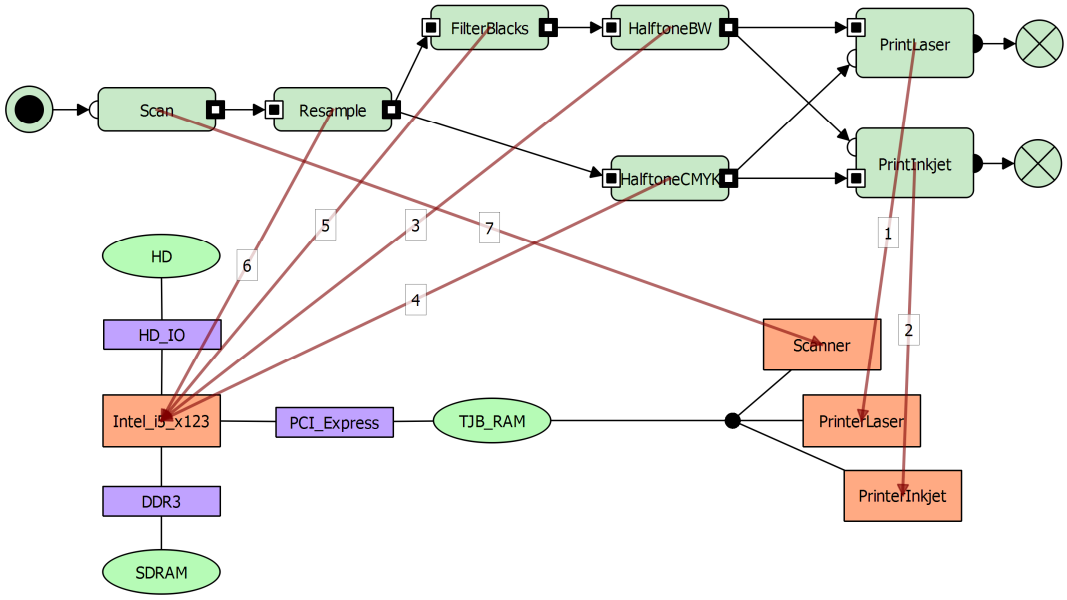

Figure 3–12. Execution links in the `corp.tonerjet.Copy_PC` mapping

Each red arrow in Figure 3–12 is an execution link, and the number associated to each link is a priority.

3.4.2. Implementations

An implementation is to execution links what tasks are to steps and what resources are to resource blocks. An implementation describes *how* a step can be mapped onto an executor block.

Because we are interested in the speed of the data path, a very 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, respectively. 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, as described in Section 3.3.1.1. Computing a step's processing speed, however, requires some more information from the user.

Recall from Section 3.3.2.2 that it is difficult to come up with a good standard unit for processor speeds: the interpretation of speed heavily depends on the operation that is being performed and on the nature of the processor. For example, an FPGA with a clock speed of 60 MHz may be able to contrast-enhance an entire line of an image in 20 clock ticks, no matter the width of the line (due to the use of *pipelining* in a typical FPGA design). The same operation on a CPU with, say, a 1 GHz clock speed may take 10 clock ticks *per pixel*. Thus, 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. DPML implementations are used to model this triangular dependency.

An 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 page, 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, the `processingDuration` is usually a function of these properties.

Listing 3–13 shows some possible implementations for the example. The implementation `Resample_CPU` 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 amount of pixels of the *largest* image, which is the input image when scaling down, or the output image when scaling up. Additionally, on average 20 clock cycles are used per pixel in the largest image.

With this information, the `processingDuration` of the step can be computed. Notice how 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 amount of pixels

  cyclesPerPixel = 20;

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

implementation Scan_Scanner performs Scan on Scanner
{
  linesPerSecond = scanner.mmPerSecond * page.in_resolutionY;
  secondsPerLine = 1 / linesPerSeconds

  processingDuration = secondsPerLine * out.length;
}

implementation FilterBlacks_x123 performs FilterBlacks on Intel_i5_x123
{
  // Uses special Intel i5 x123 instructions to handle 8*8 pixel
  // blocks at once; one block costs roughly 48 cycles

  cyclesPerPixel = 48;

  alignedWidth  = ceil(out.width  / 8) * 8;
  alignedLength = ceil(out.length / 8) * 8;
  alignedSize   = alignedWidth * alignedLength;

  processingDuration = alignedSize * cyclesPerPixel * processor.assignedSpeed;
}
```

Listing 3–13. A set of implementations that could be used by the execution links of the `corp.tonerjet.Copy_PC` mapping. Note that three relatively complex implementations were chosen, to illustrate DPML's capability of handling complex situations.

3.4.2.1. Utilizing resource inheritance

As shown in Listing 3–13, an implementation depends on both a task and an executor. However, it is well possible that a single operation can be implemented to work on a large amount of executors. For example, the same C implementation of a resample algorithm may be compiled for a large set of CPUs types, from various vendors. If the implementation does not depend on advanced instruction sets, the general behavior of this implementation is probably roughly the same across these CPU types, and the duration of performing the operation is inversely proportional to the processor's clock speed.

For this reason, it is possible to define implementations for a whole class of processors, such as "the CPUs" or "the CPUs with SSE2 support". DPML does not prescribe which classes of processors exist, but instead utilizes the concept of inheritance on resources (as described in Section 3.3.2.3) for this. An implementation that performs a task on some executor can also be mapped on a child executor. This is used in Listing 3–13, where the `Resample` task has an implementation that is expected to work the same on all resources that extend CPU. If we would later want to see how well the data path performs on the faster and more expensive `Intel_i7_y456` processor, it is not necessary to define a new implementation. This is very important, as we must make a mapping for each *combination* of applications and platforms that we want to analyze. Having to redefine implementations for all execution links in each of these mappings would be a very tedious task. Notice how we cannot do the same for `FilterBlacks_x123`, however, because it particularly depends on unique characteristics of the Intel i5 x123 processor.

3.4.2.2. Interacting with resources

The properties of mapped executor blocks can be referred to in an implementation through the `processor`, `scanner` and `printer` keywords. These keywords resolve to the properties of the resource block that the corresponding task was actually mapped on, and not to the resource mentioned in the `on` clause (which must be an ancestor of the mapped resource, however).

The processor keyword has one additional property: `processor.assignedSpeed`, which holds the amount of processing power available *for that step* (specified in whatever unit that processor's `speed` was specified). In the common situation where multiple steps run in parallel on a single CPU, the available processing power needs to be shared over all simultaneously running steps. DPML does not prescribe exactly how this sharing happens, but for computing the expected processing duration of a

particular step, one must know how much processing power is available for that particular step. This is what `assignedSpeed` is for. Note that `processor.speed` is also available, like any other resource property: this property describes the *total* speed, however, so should typically not be used in implementations.

For processors that can only execute one step simultaneously, such as FPGA subsystems, most DSPs and most GPUs, the properties `processor.speed` and `processor.assignedSpeed` always have the same value. Therefore, it is recommended practice to only use `assignedSpeed` in implementations, to avoid unexpected analysis results.

As described in Section 3.3.2.2, scanners and printers have no prescribed properties. Instead, their properties can be described in any way the user sees fit. This is illustrated in Listing 3–13, where the `Scan_Scanner` implementation utilizes the `mmPerSecond` property of Scanner in Listing 3–7.

3.4.3. Mapped steps

As shown in Listing 3–13, properties in implementations can make direct references to the pin properties of the associated task. In fact, conceptually, when a step is mapped on an executor block, the associated implementation's properties are simply merged with that of the associated task. Together, these properties define the code of a *mapped step*. All subsequent analysis tools only consider mapped steps, and not tasks or implementations individually.

It does not matter whether a property occurs in the task or in the implementation. In some situations, for instance, one could assume that a task is defined to have a certain duration, no matter the implementation. An example could be a simple delay task, or a task used only for synchronizing multiple steps in a complicated application with much parallelism. `processingDuration` is a required property, and in most cases it exists in the implementation, but it is perfectly valid for it to exist in the task instead, as shown in Listing 3–14.

```
task Wait_2s
{
  inpin  in:  Simple;
  outpin out: Simple;

  processingDuration = 2;
}
```

Listing 3–14. A task that does not need an implementation

A step whose task has all required properties can be executed without any execution link or implementation at all. Note that a task must have an execution link if it reads or writes data, because otherwise tools cannot compute which buses are used for transferring said data.

3.4.4. Implementation and mapping meta-model

The figure below shows the meta-model of mappings and implementations in DPML.

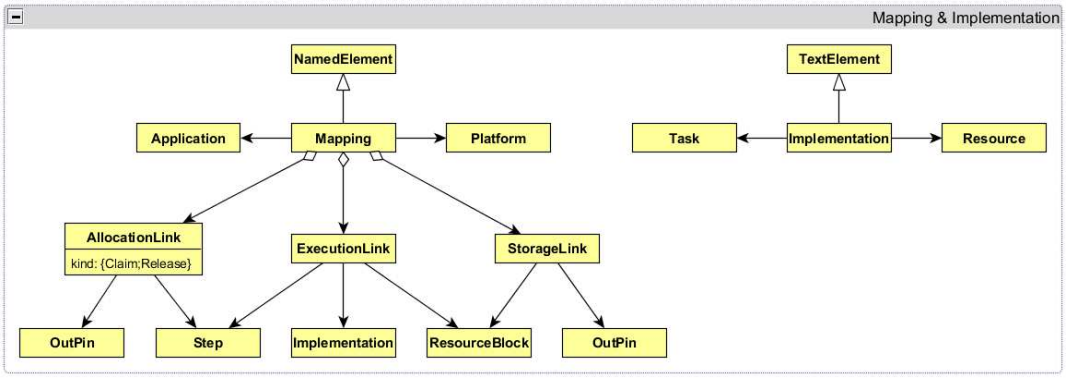

Figure 3–15. Mapping and Implementation meta-model.

3.5. The Repository

Most of the DPML elements discussed in the preceding sections can be referred to from any other number of other DPML elements. For example, a single task can be used in multiple applications, and referred to by multiple implementations. To make this cross-referencing possible, each DPML element is stored in a single small file.

When using DPML to model a variety of data paths, however, the amount of small files created quickly grows, and soon the project becomes unmanageable. Therefore, DPML includes a packaging and storage scheme called the *repository* to handle this. Few concepts in the DPML repository are new. Most notably, the repository system is very similar to how Java packages, Python modules and .NET assemblies work. The main novelty of the DPML repository therefore is, to the best of our knowledge, its application in an environment for modular modeling.

3.5.1. Files and elements

Tasks, resources, implementations, applications, platforms and mappings are all stored as a separate file. The textual components (tasks, resources and implementations) are stored as plain text files. The diagrams (applications, platforms and mappings) are stored in a custom XML format. As indicated in the previous sections, each of these elements always has a name. It is required that each named element is stored in a file with the same name. For example, a task called `Resample` must be stored in a file called `Resample.dpmlt`. This way, when the user creates a step for this task in an application, the task's properties (such as the amount of pins) can be looked up by finding and parsing a file with the correct name.

Note how this approach is very similar to, for example, Java, where a class named `Car` must be in a `Car.java` file in order to correctly compile.

3.5.2. Packages

DPML files can be structured into a hierarchy of directories, so that they can be organized in whatever way the user sees fit. The repository itself is in fact nothing but the root directory of this hierarchy. A directory with DPML files in it is called a *package*, and each package can in turn contain other packages. Package names are separated by dots instead of (back)slashes, so a directory such as *<repository_root>*`/corp/tonerjet` maps to a package called `corp.tonerjet`.

The package that a DPML element is in determines the name by which that element can be referred. For example, consider the above mentioned Resample task. If this task resides in the `corp.tonerjet` package, then an application that uses it must refer to `corp.tonerjet.Resample` instead of just `Resample`. As a single exception, if the application resides in the same package (for example, if it is called `corp.tonerjet.Copy`), then it may still refer to `Resample`. This way, moving an application and the tasks it depends on to a different package does not break the references.

This packaging and naming scheme has an important consequence: each element in DPML has a *globally unique* name. This is good, because it prevents name clashes and allows users to make models that depend on items in different packages. A common usage scenario is the creation of one or more utility packages, in which really common resources and tasks are stored. The availability of such common items may significantly speed up the modeling process and avoid doing double work.

3.5.3. Sharing the repository

The DPML repository is stored simply as a directory tree with text files and XML files; no explicit support was included for the repository, or parts of it, to be shared among different engineers. However, since the DPML items are all small text files, existing revision control software used at the research and development site may be ideally suited for such a sharing purpose. Combined with the concept of a few well managed utility packages, such a shared repository may help sharing design knowledge and choices between different projects.

For example, if engineers of a project discover that the I/O performance of a particular hard drive is structurally lower than advertised, they can update the properties of that DPML resource, document the change in a source code comment, and check it into revision control. Other teams that consider using the same hard drive may find that it already exists in the repository, and use that resource instead of crafting their own with the (incorrectly) advertised I/O performance.

4. Conclusion

Our goal is to enable architects to model and analyze data path architecture options early on in the design process. In addition, architects should be able to discover the bottlenecks in a data path design, and compare different design alternatives.

This enables making the data path design process shorter and more structured. Additionally, this helps to reduce the impact of changes in requirements and in the environment, because the result of foreseen changes can be analyzed in advance, and in the case of unforeseen changes, the set of possible alternatives can be quickly compared.

The goals mentioned above have been met by means of a custom tool chain. This tool chain is based on DPML, a domain-specific language for modeling data path designs. DPML is expressive, powerful and easy to use and it comes with a custom editor that is user friendly and future-proof.

DPML consists of three components: the application, the platform and the mapping. The application is a directed graph of steps, which specifies a use case for the printer. The steps have properties defined in tasks. The platform is an undirected graph of resource blocks, whose properties are defined in resources. The mapping, finally, is a labeled directed graph between an application and a mapping. Storage links define which data is stored on which memory resource blocks. Allocation links define which steps reserve memory for this data. Execution links define which steps run on which executor resource blocks. An execution link is tied to an implementation, which defines execution properties of that particular task on that particular resource.

The four design goals for DPML stated in the beginning of Section 3 have been met as follows:

- DPML has a focus on measuring **speed**. The ability to specify precise step behavior with respect to the content of the images being manipulated has been omitted, but the data sizes of images being manipulated and transferred play a dominant role. Data sizes can be used implicitly to compute transfer times, but can also be used directly in the model for specifying the processing time of each step. Additionally, many facilities have been included to specify speed-impacting peculiarities of particular resources or operations in high detail.

- DPML is **expressive** and **flexible**. By incorporating a textual expression language (JavaScript) capable of all common mathematical operations, relationships between various properties can be easily expressed. Additionally, because it is possible to add custom properties to any element, many currently unforeseen concepts may be automatically supported by DPML. Finally, because the textual elements are specified in a simple custom language and stored in small text files, it is easy to make tools that can deal with additional special properties or additional syntax, without requiring changes to a custom modeling environment.

- DPML is close to the **problem domain**, printer data paths. The two most visible examples of this are the concept of pages flowing through the steps of the application and the printer and scanner resource types. More important, however, are the features that DPML does *not* have that would make it a more generic specification language, such as support for caches, the ability to fully specify the behavior of steps, the possibility to specify real-time deadlines, or the ability to fine-tune a task scheduler. The omission of 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.

- DPML is **modular** because all elements are stored in separate little files, and any element can be referred to from multiple other elements. Additionally, because of the separation of the DPML elements along the Y-chart, a single use case can be used to analyze many hardware designs and a single hardware design can be used to analyze many use cases. Together, these two features help preventing double work and encourage information sharing across projects and teams.

DPML models can be converted to DPML Compact, a representation of data path models that is simple in structure yet captures all models that can be expressed with DPML. Because of their simplicity, DPML Compact models are an ideal starting point for connecting to analysis tools like CPN Tools or Uppaal.

The tool chain includes a simulator which takes DPML Compact models as its input and simulates the behavior of the modeled data path design. The simulator produces trace files that can be understood by ResVis, an existing tool that is ideally suited for visualizing the results of this simulation.

This tool chain enables engineers and architects to model and analyze data path architectures. Additionally, because of the expressive nature of DPML, the simple structure of all tools and the maintainability requirements that guided their implementation, it is feasible and worthwhile to extend the tool chain so that more precise analyses can be performed.

Therefore, the results presented in this paper enable architects to make informed decisions about data path architecture alternatives and to take the impact of possible future changes in requirements and the environment into account. This, in turn, enables to make the data path design and development processes more structured, more controlled, and more flexible.

The results also open the door for additional uses of data path models. By extending DPML and the associated tools, highly detailed simulations of how a data path will behave in reality may be performed. Besides that, the simulator may be leveraged to serve as a part of a software-in-the-loop testing solution. This way, the software that controls the data path may be tested before the data path is fully implemented. Finally, computer-aided design space exploration of data path architectures may be envisioned, helping architects to choose the best design even faster and better.

References

[1] G. Behrmann et al. Uppaal 4.0. In *Proc. QEST*, pages 125-126. IEEE CS Press, 2006.

[2] T. Basten et al. Model-Driven Design-Space Exploration for Embedded Systems: The Octopus Toolset. In *T. Margaria et al., editors, 4th Int. Symp. on Leveraging Applications of Formal Methods, Verification and Validation, ISoLA 2010, Proceedings*, Part I, pages 90-105. LNCS 6415. Springer, Heidelberg, 2010.

[3] M.H.H. Brassé, S.P.R.C. de Smet. Data path design and image quality aspects of the next generation multifunctional printer. Image Quality and System Performance V. Edited by Farnand, Susan P.; Gaykema, Frans. *Proceedings of the SPIE*, Volume 6808, pages 68080V.1-68080V.11 (2008).

[4] A.W. Brekling et al.. Models and Formal Verification of Multiprocessor System-on-Chips. J. Log. Algebr. Program., 77(1-2):1-19, 2008.

[5] CoFluent Design. CoFluent Studio. http://www.cofluentdesign.com/.

[6] A. Davare et al. A Next-Generation Design Framework for Platform-based Design. In *Proc. DVCon 2007*, February 2007.

[7] A. David et al. Model-based Framework for Schedulability Analysis Using Uppaal 4.1. Model-based Design for Embedded Systems, pages 121-143. Taylor & Francis, 2009.

[8] K. Jensen et al. Coloured Petri Nets and CPN Tools for Modelling and Validation of Concurrent Systems. *STTT*, 9(3-4), 2007.

[9] B. Kienhuis et al. An Approach for Quantitative Analysis of Application-specific Dataflow Architectures. In *Proc. ASAP 1997*, pages 338-349. IEEE, 1997.

[10] A. Ledeczi et al. Modeling methodology for integrated simulation of embedded systems. *ACM Trans. Model. Comput. Simul.*, 13(1):82-103, 2003.

[11] MLDesign Technologies. MLDesigner. http://www.mldesigner.com/.

[12] Nokia Corporation. Qt - A cross-platform application and UI framework. http://qt.nokia.com/products.

[13] A.D. Pimentel. The Artemis Workbench for System-Level Performance Evaluation of Embedded Systems. *Intl J. Embedded Systems*, 3(3):181-196, 2008.

[14] I. Sander and A. Jantsch. System Modeling and Transformational Design Refinement in ForSyDe. IEEE T. Comput.-Aid. Design, 23(1):17-32, 2004.

[15] K. Schindler. Measurement data visualization and performance visualization. Internship report, Dept. of Mathematics and Computer Science, Eindhoven University of Technology, 2008.

[16] S. Stuijk, M. Geilen, and T. Basten. SDF3: SDF For Free. In Application of Concurrency to System Design, 6th International Conference, ACSD 2006, pages 276–278. IEEE CS Press, June 2006.

[17] B.D. Theelen et al. Software/Hardware Engineering with the Parallel Object-Oriented Specification Language. Proc. Memocode 2007, pages 139-148. IEEE, 2007.

[18] I. Viskic et al. Design Exploration and Automatic Generation of MPSoC Platform TLMs from Kahn Process Network Applications. Proc. LCTES, pages 77-84. ACM, 2010.