UvA-DARE (Digital Academic Repository)

Tools and techniques for efficient system-level design space exploration
Thompson, M.

Link to publication

Citation for published version (APA):
Thompson, M. (2012). Tools and techniques for efficient system-level design space exploration

General rights
It is not permitted to download or to forward/distribute the text or part of it without the consent of the author(s) and/or copyright holder(s), other than for strictly personal, individual use, unless the work is under an open content license (like Creative Commons).

Disclaimer/Complaints regulations
If you believe that digital publication of certain material infringes any of your rights or (privacy) interests, please let the Library know, stating your reasons. In case of a legitimate complaint, the Library will make the material inaccessible and/or remove it from the website. Please Ask the Library: http://uba.uva.nl/en/contact, or a letter to: Library of the University of Amsterdam, Secretariat, Singel 425, 1012 WP Amsterdam, The Netherlands. You will be contacted as soon as possible.

UvA-DARE is a service provided by the library of the University of Amsterdam (http://dare.uva.nl)
Chapter 3

Sesame: modeling and simulation

3.1 Introduction

As was shown in the previous chapter, the Sesame tool is used within Daedalus for efficient system-level modeling and simulation. Most of the topics discussed in the second part of this thesis (Techniques: Chapters 4, 5 and 6) either use Sesame directly, or with the modifications proposed in those chapters. In the current chapter we will give an overview of the Sesame environment to give the reader a general insight into the methodology and scope of Sesame. Although Sesame is the tool of choice in the context of the Daedalus toolflow, it may be equally useful in other contexts where system-level modeling and simulation is required. Therefore, in this chapter, Sesame will be presented as a generic modeling and simulation tool, but where appropriate, its connection to the Daedalus environment will be discussed.

The structure of this chapter is as follows. In the next section a general overview of the Sesame environment will be presented. A Sesame model consists of three parts: the application model, the architecture model and the mapping model: shown in Figure 3.2 as the top, bottom and middle model layer respectively. The discussion in the next section will follow the natural order from modeling the functional behavior of the system to modeling the system platform and hardware aspects with respect to non-functional behavior. Section 3.3 revisits these topics with a focus on the implementation details of different parts of the model in Sesame. Some modeling examples will be given to make the previously discussed topics more concrete. The final section discusses issues related to the periphery of Sesame: methods for efficient specification of the design space (Section 3.4) and interpretation of simulation results (Section 3.5).

3.2 Overview

One of the main design principles of the Sesame environment is to apply separation of concerns where possible. In this way, complicated problems are decomposed in smaller, easier to solve sub-problems, which (combined) still represent the original complicated problem. The most prominent separation of concerns in Sesame is the separation of functionality and...
implementation, which is also proposed by the Y-Chart design principle ([54]). Sesame adheres to this principle by providing separate application and architecture models: the former captures the functional behavior of the system while the latter captures non-functional behavior. The non-functional behavior modeled by Sesame principally concerns system performance, but the model can be extended to account for additional objectives such as energy [81] or cost. The Y-Chart principle is schematically depicted in Figure 3.1 and consists of three stages: 1) application and architecture model creation, 2) mapping application to architecture and 3) model evaluation and feedback. The application model describes the functional behavior of an application in an architecture-independent way and it does not contain any architectural specifics such as the resource or performance constraints of architectural components. To obtain a rough estimate of its performance requirements, the application model can typically be studied independently from the architecture model using traditional software analysis tools such as software profilers. However, for the purpose of more detailed analysis, an architecture model is created that models for example system performance, power or cost. Sesame uses trace-driven cosimulation where the application model "drives" the architecture model by providing it with dynamic application information. A trace consists of events, and each event describes a single atomic action by the application model. The information contained in an event is typically a high-level description of a communication or computation action of the application: detailed functional information is omitted. For example, a data communication is described by its source, target and data size, but the actual values of the data are not included because they are not needed in Sesame's high-level non-functional architecture models. In the remainder of this section we will discuss how the three Y-Chart stages are represented in Sesame and give further details of the
interaction between the application and architecture models. Section 3.3 will give more
details towards the implementation of Sesame.

![Diagram of the Sesame model](image)

Figure 3.2: Overview of a Sesame model: the application, architecture and virtual layer

### 3.2.1 The application model

In Sesame we use Kahn Process Networks (KPN) as the preferred Model of Computation
(MoC) to represent the application. This choice is motivated by the application domain that
is targeted by Daedalus: multimedia and signal processing applications are often data-flow
dominated and can be thought of as streams of data passing through a network of actors
performing computations on individual data items (which may represent, in the example
of an imaging application, pixels, lines, pixel blocks or image frames). A KPN is defined
as a network of concurrently executing processes without access to shared memory, which
communicate exclusively over Kahn channels. Kahn Channels are one-directional FIFO
buffers of unbounded capacity without restrictions on the type of data that can be sent. In
the top section of Figure 3.2, an example is given of a simple KPN with 4 processes and 4
channels. Internally, Kahn processes may be defined in any high-level language as long as
the Kahn semantics are observed.
The KPN MoC is defined mainly by the semantics that are enforced on the channel communication. Processes are allowed to read to and write from any channel at any time. Write operations always succeed, since the channel buffers are of infinite capacity. However, reading from an empty channel causes a process to stall (“block”) until data is written to this channel. A further condition is that there is no “test” operator to check data availability and therefore process execution is independent of the state of the channel. There are no restrictions on what a Kahn process is allowed to do internally, except (as mentioned before) that it is not allowed to access memory that is shared with other processes: all inter-process communication has to occur explicitly over Kahn channels. Similar to the Synchronous Data Flow MoC (SDF), KPNs fall in the untimed MoC category: there is no concept of time passing, neither in terms of clock cycles nor in (simulated) time.

The result of the above semantics is that the KPN MoC is deterministic: the order of the tokens that are communicated over the channels does not depend on the execution order or execution time of individual processes. Determinism is a highly desirable property for the kind of models that we are considering: the system model (the combination of application and architecture) will produce the same output (for given input) regardless of the specific process scheduling or architectural (timing) characteristics. It also guarantees the validity of the event traces between the independently executing application and architecture models. Although determinism guarantees the functional behavior of the system, no assertions can be made yet about the system’s non-functional behavior. For this purpose we need either an implementation on a real system or an architectural simulation that captures certain non-functional behavior.

3.2.2 The architecture model

The architecture model in Sesame (see the bottom of the three layers in Figure 3.2) is responsible for modeling latencies associated with the physical properties of the system. The Sesame architecture model is specified using the Pearl discrete event simulation language. The Pearl language was designed from the ground up to efficiently model multi-processor heterogeneous computer architectures at a high level of abstraction. On the one hand it achieves very good runtimes for entire system-level simulations running an application with a representative input workload. On the other hand, it sports an easy and compact concurrent programming style with powerful semantics that enables designers to express complex interactions between model components with relative ease. This means that complete system level models can be defined while the model development time remains modest, which is especially important in the early stages of development as architectural design decisions may be taken at the highest level and models may frequently need to be changed to test and evaluate new designs. The choice of programming language is therefore more than a trivial choice, since it can directly affect the overall design time of the system. In the following we discuss some of the main language features of Pearl. In Section 3.3.3, some more details and a simple model example will be given.

The Pearl programming language is characterized by two main features: 1) an object oriented approach for defining model components and 2) integrated primitives for commu-
3.2. OVERVIEW

and synchronization between model components. The functionality and behavior of model components is specified as a class, similar to a class definition in object oriented languages such as Java or C++. Also similar to object oriented languages is that each class has a private namespace for variable and functions in order to avoid variable name clashes. A light-weight class hierarchy scheme is available to group related classes into a family with a common superclass. The Pearl syntax will be familiar to C users, but the language has been designed to include only those features that are necessary for architecture modeling. In particular this means that there is no support for advanced data structures (other than classes for model components), no pointer support, and only a limited set of basic data types consisting of integers, floats and strings and one-dimensional arrays of these basic data types. In rare cases where a programmer needs more advance language features, it is possible to incorporate C functions which will be linked into the final simulation executable. At simulation time, classes can be instantiated as simulation components (also referred to as modules). Class instantiation is done automatically by the simulator prior to the start of the actual simulation according to an XML-based specification which is defined by the user (see Section 3.3.1). Since traditionally the components in a hardware system are fixed, Pearl does not support instantiation of classes after the initialization of the simulator. This assumption does not generally hold anymore with the advent of dynamically reconfigurable hardware platforms such as FPGAs. Indeed this topic is addressed in Chapter 6 where we show how Pearl can still be used to model this emerging type of systems.

The main difference with traditional object-oriented languages, however, is that each module maintains its own execution context, or thread, so that modules can easily express the naturally occurring parallelism between components in hardware. Modules can not modify data items in each other’s address space. Instead, Pearl has Remote Procedure Call (RPC)-like primitives for communication and synchronization between modules. For example, the following synchronous function call statement:

\texttt{moduleref ! functionname(parameters);}

calls a method with the name \texttt{functionname} on a module pointed to by \texttt{moduleref} with the given \texttt{parameters}. The Pearl runtime system translates the call to a message which is sent to the remote module. The calling module will now be halted until the remote module has completed the function (using a \texttt{reply()} statement, which can optionally be used to return a value to the calling module). Note that in addition to this synchronous method call (that halts the calling module), there is also an asynchronous method call (that gives control immediately back to the caller) using the operator “!!”. For both synchronous and asynchronous calls, the function will execute in the context of the remote module \texttt{and} at a time that is specified by the remote module (note the difference with a method call in a normal Object Oriented language). Therefore, the runtime system will store (completely transparent to the user) all function calls issued on a certain module in a dedicated message queue. The remote module is itself in charge of accepting calls to certain functions by issuing a matching \texttt{block} statement:

\texttt{block(functionname);}
CHAPTER 3. SESAME: MODELING AND SIMULATION

The result of the \texttt{block} statement is that the remote module takes the function call message from its queue and executes it in its own thread context. When this function call exits, the issuing module is unblocked and both modules continue with the next statement in their code listing. If a module issues a \texttt{block(functionname)} statement before the \texttt{functionname} has been called, then the module will block until such a call is made.

The final important Pearl statement is used to indicate that a module is busy for a given number of clock cycles (e.g., a processor component is busy executing some computational kernel or a bus is busy transferring some data):

\begin{verbatim}
block(t);  
\end{verbatim}

During the specified amount of \texttt{time} the module is suspended and can not perform other actions such as remote function calls; it will resume after \texttt{time} clock cycles have passed. The simulation runtime will execute without priority and non-preemptively any runnable module with simulated concurrency. Modules are runnable until they yield by blocking on remote function calls or with \texttt{block}. The Pearl simulation runtime maintains simulation time using the globally shared clock. Note that a system with multiple clock domains can be modeled by relative scaling of latencies (small errors introduced by scaling are insignificant at Sesame’s high level of abstraction). When there are no more runnable modules, then the discrete-event simulation engine will increment global clock to the earliest resume-time for a suspended module and resume execution of that module. Valid termination occurs when either all modules have finished execution, or when all modules are blocking in unmatched communication primitives: a remote function call primitive without a matching \texttt{block}, or vice versa.

In Pearl, the remote procedure call primitives perform -- in a completely transparent way -- relatively complex communication and synchronization transactions between modules. Furthermore, the compact syntax of the primitives ensures that simulation code in Pearl is easy to write, which is an essential property for early model development. As we will see in the example in Section 3.3.3, Pearl code is also easy to read, since the combination of RPC primitive and function name completely determines the state a module at any given time. Code modularity (and thereby code reuse) is further promoted by the use of classes to define simulation components. The set of remotely callable functions in a class acts as an explicit module interface: a module can be easily be replaced by another module as long as it implements the same functions. All in all, Pearl is a suitable language for the creation of simulation models that support design decisions in the early stages of MPSoC design where models typically have to be created and modified in short time frames. We shortly refer to the work in [104] where we transferred the functionality of the compact Pearl communication primitives to the more widely-used and standardized SystemC modeling language. However, this approach suffered from performance limitations of the reference SystemC implementation, thus favoring the use of the custom Pearl language within Sesame to improve model execution speed.
3.2. OVERVIEW

3.2.3 Mapping

Due to the separation of application and architecture, an explicit interface is needed to connect the two models. Sesame performs trace driven simulation where the events collected in the untimed application model are streamed to the timed domain of the architecture model. The events are abstract representations of the actions performed by the application model and are initially chosen to represent only coarse grain events. At this coarse grain level of abstraction only channel communication and computation of large kernels or functions are represented by events. In Sesame, these event traces are not immediately consumed by the architecture model, but instead they pass through a so-called "virtual layer" (sometimes called mapping layer). The virtual layer serves multiple functions towards mapping, synchronization and scheduling of events before they are passed on to the architecture model. The virtual layer exists in the same timed domain as the architecture model and is in fact part of the same simulation model, however, we treat it here as a separate entity from the architecture layer as its functionality is distinct from the architecture model. The virtual layer is composed of virtual processors and virtual channels that are connected according to the same topology as the application model (see Figure 3.2). For this reason the virtual layer can initially be generated automatically (this process will be further explained in Section 3.3.4). The virtual processors read the event traces that are generated by the application model and forward them (according to the mapping specification) onto architectural model components such as processors. Before forwarding events, however, the virtual layer takes care of modeling several important issues with regard to synchronization of the events as well as some issues that are optional for certain types of models. In the following we will shortly discuss the different functions of the virtual layer.

**Synchronization modeling**

According to the Kahn MoC, communication takes place in a world where there is neither time, nor a limitation on the channel size. Of course, in any practical implementation of a system this assumption does not hold and (apart from physical wire and switching delays) the speed of communication is limited by the storage capacity of the transport medium (bus width, packet size or memory size, etc). This limited capacity may lead to problems when pairs of producing and consuming nodes are not producing and consuming data at the same rate. For example consider the case where a receiving node consumes packets slower than the production rate of packets; eventually the (limited) buffer capacity will fill up and the producing node will be forced to wait until storage space becomes available. Similarly, a consuming node may be blocked on an empty channel when the production rate is slower than the rate of consumption. Within the context of Sesame, the time spent waiting for empty or completely saturated communication channels is referred to as synchronization latency and it is modeled by the virtual layer. Note that latencies associated with architecture specific parameters such as bus width, throughput or latency are modeled in the architecture model, as will be shown in the next subsection.

The virtual processors and virtual channels can be seen as the real-world representatives
of the Kahn applications and channels: they do in fact model timing consequences of waiting for data (when a channel is empty) or waiting for free space (when a channel is full). In the case of a virtual processor this works as follows: for each communication event (RD or WR) the virtual processor checks whether the data is available or whether there is sufficient storage space for a write event to complete. If necessary, the virtual processor will halt until the respective condition is met. The amount of time the virtual processor is halting is referred to as a synchronization latency. When the condition is met, only then is the communication event forwarded to the architecture model to model physical latencies that may be associated with communicating data over a certain medium (communication network or memories for example). Note that the synchronization latency indirectly includes architectural latencies (which are modeled in the architecture layer). For example, when a write is blocked on a full channel, then the total write synchronization latency includes both waiting for the initiation of the corresponding read event and the latency modeled by the architecture model to complete the read (including e.g., bus and memory latencies).

This reveals a second case of separation of concerns in Sesame: synchronization latencies are modeled separately from the hardware communication latencies. This turns out to be particularly useful when the synchronization behavior of certain tasks needs to be changed, as we will see in Section 3.3.4.

Deadlock prevention

For completeness, we list the prevention of deadlock as one of the tasks of the virtual layer, although this is actually a side-effect of the synchronization modeling in the previous chapter. Deadlocks could occur by chance when two dependent application processes are mapped onto the same processor, depending on the order of the trace events. Such deadlocks need to be prevented, since a deadlocked model in Sesame satisfies the model termination condition and thus no useful information can be derived from a deadlocked simulation run. For example in Figure 3.2, assume process A writes to process B, but the communication buffer in the virtual layer is full. If the write event from A had already been sent to the architecture model, then this write event would block processor P0 and deadlock would occur, since the read event from process B (which would free the required buffer space) has to execute on P0 too. The synchronization mechanism from the previous section resolves this problem by delaying forwarding the write event to P0 until sufficient space is available in the buffer.

Model Refinement

An important consideration in Sesame is to support mixed-level modeling and gradual model refinement. In a typical design case study it may be necessary to model some parts of a system in more detail, for example because their effects on global system performance can not be captured accurately by the high-level model. Sesame enables a designer to start with a high level model and gradually refine certain parts of the model as needed. The resulting model mixed model contains components specified at different layers of abstraction, but still runs as a single simulation entity. Moreover, by gradually refining more and more
3.2. OVERVIEW

Components towards a lower level of abstraction (specifying ever more details about the system), the model will come closer to a level of specification which is a sufficient starting point for synthesis using traditional methods. As we explained in the previous chapter, Daedalus is more geared towards system synthesis based on high-level specifications, but the refinement to synthesizable specification is still an often used design trajectory.

However, when architecture model components are refined, the level of granularity of the events produced by the application model may not be sufficient to "drive" the underlying refined architecture model components. Because of the relatively high abstraction level of the atomic application events (a sequence of Rd, Wr, Ex-events which is executed in strict trace order), information about the application model is lost that can be needed for the refined architecture components, eg: control flow information, event dependencies, etc. An example is an image processing component that works on frames, but the refined architecture component works on pixel lines and has only local memory to contain a few pixel lines. The processing component should now read the frame data using multiple data transfers, which in turn need multiple synchronization events. Such information was not present in the original application event trace. One option is to refine the application model together with the architecture model to generate events containing this additional information, but this would break the separation of application and architecture.

To resolve this problem (while maintaining the benefits of separate application and architecture models), Sesame provides dataflow based trace transformation techniques in the mapping layer. For this purpose, the Virtual Processors in the mapping layer are extended with automatically synthesized, modified versions of Synchronous Dataflow (SDF) and Integer Data Flow (IDF) graphs which act as executable, yet abstract representations of the original application processes. The nodes in these data flow graphs represent synchronization events such as Sr (Signal Room), Sd (Signal Data), Cr (Check Room), Cd (Check Data) or architectural events such as Ld (Load data) or St (Store data). While synchronization nodes

![Figure 3.3: Virtual processors with refined synchronization behavior](image-url)
are typically connected to each other to express dependencies and available parallelism. Ld and St nodes directly connect to architectural model components (Figure 3.3). When a Ld or St node fires, events are forwarded to the architecture model, where the latency of the event is modeled, which in turn completes the firing action of the node. Note that this so-called token-exchange mechanism between the virtual and architecture layer executes in the timed Pearl simulation domain, which yields timed dataflow models, resulting (in contrast to pure SDF-models) in a semi-static scheduling. In order to support more complex refinements, e.g. for Kahn processes containing loops and conditional code within Kahn processes, special IDF models have been defined.

As an example we use the default synchronization behavior that was explained in the previous section. It is in fact also a form of trace refinement: the Rd and Wr events have been refined in a sequence of smaller communication and synchronization events:

\[ Rd : Cd \rightarrow Ld \rightarrow Sr \]

\[ Wr : Cr \rightarrow St \rightarrow Sd \]

Figure 3.3 shows the SDF corresponding to this refinement for two virtual processors A and B. Note the dependencies between the synchronization events and the initial token that is required for the first Cr (check room) event. The token exchange mechanism (consisting of the refined trace events) with the architecture model is depicted as a line that connects the Ld and St SDF nodes to model components in the architecture layer. This type of synchronization behavior is currently the default for any Virtual Processor in Sesame and further model refinement is optional.

To conclude, we can say that the combination of SDF and IDF refinement models support a smooth transition between abstraction levels both for communication (and synchronization) and execution events while keeping the application model unchanged. In this way refined architecture models can be used that capture intra-task level parallelism, various communication policies as well as pipelined processor components. For more details and examples we refer to [78].

**Scheduling**

As explained previously, virtual processors are the “timed representatives” of application processes and the application mapping definition determines to which architectural component the virtual processors forward trace events. Simultaneous events from different processes will compete for resources when they have been mapped onto the same architectural component (e.g., Processes A and B in Figure 3.2). In the standard non-refined Sesame system model, virtual processors (after modeling synchronization (Section 3.2.3) directly forward events to the architectural component. As a result of Pearl’s inter-process communication semantics based on message passing, simultaneous events will be queued at the processor and architectural timing consequences are modeled in a first-come first-serve (FCFS) manner by the processor component. The FCFS scheduling policy for application events is a suitable model for systems supporting dynamic scheduling of tasks on the execution units. This may for example be a processor running an operating system that implements pre-emptive threads (e.g., POSIX pthreads), user threads (light weight processes), OS-level
processes or some kind of job scheduling middleware. Note that the particular scheduling technology or scheduling unit (thread, process, etc.) is irrelevant at the high level of abstraction of the unrefined Sesame model: we simply assume the resource will schedule a task as soon as all earlier tasks have been processed.

There are cases, however, where the scheduling behavior of architectural resources differs significantly. For this purpose, a scheduler model component has been defined that can be used in the virtual layer to enforce specific user defined scheduling policies. The scheduler consults an external policy component to identify when tasks are to be forwarded to the resource. In this way a wide range of different scheduling policies can be implemented; examples include fixed predefined (static) schedules, time slicing or priority scheduling. The only limitation of the scheduler is that it can not split the application events, e.g. in order to implement a pre-emptive scheduling after an event has already been forwarded to the architecture. Therefore, application events have to be considered atomic, unless they have been explicitly refined (Section 3.2.3). The scheduling policies are defined as separate components to promote reusability and to separate the event scheduling from policy definitions. Schedulers in Sesame will be further discussed in Chapter 5, where they are used, for example, to model multi-application workloads.

3.3 Implementation Aspects

In the previous sections we showed Sesame’s distinct application and architecture model layers, their important characteristics and their interaction. In this section we focus more on the implementation aspects of the models and we again follow the structure of the layers in a Sesame model: application model, architecture model and finally the one connecting both: the virtual (or mapping) layer. But first we discuss Sesame’s model description language called YML (Y-chart modeling language), which is used to describe both the structure (and topology) of the various models and to specify the initialization values of each component.

3.3.1 Model specification

YML is a specialized dialect of the more commonly known XML (Extensible Markup Language). YML has the advantage of being both a machine readable as well as human readable format (at least for smaller specifications). Each YML element defines a component or topological relationship in the model. For example we use elements with tag-names node, network or link to describe respectively a component, a group of components or a relationship between components. XML attributes specify the additional required details for each type of element. All element types and their optional and obligatory attributes have been formally defined in a schema (XSD) specification. By using the schema, it is easy to use existing parsing solutions, which not only simplifies the implementation of the Sesame tools, but also greatly enhances the possibility to exchange specifications with external tools which may have been developed by third parties.
XML is defined as a hierarchical specification language where certain elements (according to the relevant schema) may contain other elements, which in turn contain other elements, etc. This way, what is essentially a tree-like structure is defined, which is very suitable to describe applications or architectural topologies. In Sesame’s YML, the starting element of a model is always a network element in which all other model components are contained. As an example, see Figures 3.4 and 3.6, where a very compact application and architecture model is specified. The attributes of the root-level network specify the URI for the YML schema definition, a name and the type (class) of the network to indicate whether this is an application (KPN) or architecture model specification (net). Additional property elements describe additional information such as the location of the compiled application code (Fig. 3.4 line 8-10, or Fig. 3.6 line 14-15) which will be used respectively by the application or architecture simulation runtime. In the following we will describe the implementation-level details of model components at each of the Sesame layers and their specification in the corresponding YML file.

3.3.2 The application model

```xml
<?xml version="1.0" encoding="UTF-8"?>
<network xmlns="http://sesame.im.sourceforge.net/YML"
    name="application" class="KPN">
    <property name="library" value="libjpeg.so"/>
    ...
    <node name="ND_2" class="CPP_Process">
        <property name="class" value="ND_2"/>
        <property name="header" value="src/m_func.h"/>
        <property name="source" value="src/_2.cpp"/>
        <port name="IO_1" dir="in"/>
        <property name="type" value="struct T1locks"/>
    </port>
    <port name="IO_1" dir="out"/>
    <property name="type" value="struct T1locks"/>
</node>

<node name="ND_3" class="CPP_Process">
    ...
</node>

<link innode="ND_2" import="IO_1" outnode="ND_3" outport="IO_1"/>
<link innode="ND_3" import="IO_1" outnode="ND_4" outport="IO_1"/>
...
<property name="operation:op_initVideoIn" value="0"/>
<property name="operation:op_mainVideoIn" value="1"/>
<property name="operation:op_mainVideoOut" value="2"/>
...
</network>
```

Figure 3.4: Sample application XML

Figure 3.4 lists part of the YML specification for a MJPEG application that has been automatically derived from sequential code by the Daedalus PNGen tool. The node elements represent Kahn processes of the class `CPP_Process` (as indeed we will see shortly they
3.3. IMPLEMENTATION ASPECTS

are implemented in C or C++, and a name attribute. The port elements (line 11 and 14) indicates that the process contains an in-bound port and an outgoing port. The ports form the anchor points for Kahn channels which communicate data of the type as specified by the port (in this case 8x8 pixel macroblocks: struct TBlocks). The link property (line 20) connects the out-port of process ND_2 with an in-port of process ND_3, thus connecting the Kahn channel between the two processes. Figure 3.5 lists the C++ implementation of the

```cpp
#include "ND_2.h"

ND_2::ND_2(Id n, ND_2_Ports *ports) : ND_2_Base(n, ports) {}

void ND_2::main() {
    for (int c0 = ceil(0); c0 <= floor(7); c0 += 1) {
        for (int c1 = ceil(0); c1 <= floor(15); c1 += 1) {
            for (int c2 = ceil(0); c2 <= floor(7); c2 += 1) {
                ports->in_0[0].read( in_0ND_2 );
            }
        }
    }
}
```

Figure 3.5: Sample application process code

ND_2 process. Its regular structure of affine nested loops and slightly odd variable names is due to the fact that this code generated by PNGen. In general, a Kahn process can have many in- and outgoing channels and there is no restriction on the number or ordering of reads and writes, the number of produced execute events, or the naming conventions for nodes and ports. Channels are accessible from the code by means of the ports structure which is inherited from the (automatically generated) base class ND_2_Base. The runtime system will automatically instantiate and initialize the processes and channels before it proceeds to execute the process network. The ports connect processes by means of Kahn channels according the topology defined by the link elements in the YML. Input ports and output ports have, respectively, read() and write() functions defined on them, where a read is blocking if the channel is empty and the write is always non-blocking due to infinite capacity of the channels.

Note that the process network is a functional application: in the case of ND_2 the DCT function operates on real data and the result of the given KPN network is a sequence of compressed MJPEG images. The DCT function (line 14) is called on the input macroblock (line 12) and writes the result to the next process which will perform quantization (line 17). Important to note is the execute statement at line 15 which has the sole purpose of generating a trace event (which has identifier 2 as specified in the YML (Fig. 3.4 line 25). The port
readQ and writeQ functions generate trace events as a side-effect of their communication
actions, so these do not have to be specified explicitly. The trace events from each process
are captured as an ordered stream of events which is forwarded first to the components
in the virtual layer and then to the architecture model, where timing consequences of the
events are modeled.

3.3.3 The architecture model

```xml
<?xml version="1.0" encoding="UTF-8"?>
<network xmlns="http://sesameim.sourceforge.net/YML"
    name="architecture" class="net">
  <node name="processorX" class="pearl.object">
    <property name="class" value="processor"/>
    <property name="header" value="processor_h.l"/>
    <property name="source" value="processor.c"/>
    <property name="init" value="out0 = [0|32,0,0,0,154,1,45,0,1,154]"/>
    <port name="out0" dir="out" ...
    <property name="template" value="hvproc">"/>
  </node>
  <node name="vproc" class="pearl.object">
    <property name="class" value="vproc"/>
    <property name="header" value="vproc.h"/>
    <property name="source" value="vproc.c"/>
    <property name="init" value="trace, #&lt;vchannel1&gt;">"
    <port name="out0" dir="out" ...
    <port name="trace" dir="out" ...
    <port name="channels" dir="out" ...
    <link innode="hvproc" import="out0" outnode="chief" outport="channels"/>
  </node>
  <node name="bus" class="pearl.object">
    <property name="class" value="bus"/>
    <property name="header" value="bus.h"/>
    <property name="source" value="bus.c"/>
    <property name="init" value="out0"/>
    <port name="out0" dir="out" ...
  </node>
  ...
</network>
```

Figure 3.6: Architecture YML

In Figure 3.6 we list a part of the YML specification for an architecture description of
a multi-processor system connected to a shared memory. The node elements in the YML
specify which simulation components will be instantiated in the Pearl simulation language.
In Figure 3.7 a slightly simplified version of the Pearl code for each of the main components
is given. In the YML specification, the element on line 4 starts the definition of processorX
and subsequent lines list its source code files. The init string on line 8 gives the initial-
3.3. IMPLEMENTATION ASPECTS

After proper initialization, the variables `b` and `operations` in the processor object will be initialized (Fig. 3.7 line 3-4). The first of these variables is a reference with the name `b` that points to a `bus` object. According to the `init` string (Fig. 3.6), this reference is initialized to whatever the `output` output port is connected. The `link` element specifies that it connects to an input port on the bus object. Thus, a synchronous or asynchronous Pearl method call on `b` means a call to the bus object defined on line 26 of the YML. This is a clear example that the model topology is specified separately (in the YML file) from the model behavior (in the Pearl source files). This facilitates model component reuse and makes it easier to re-run experiments (without re-compilation) by simply changing the parameters in the YML files. Lines 10-24 in the YML define a virtual processor template, from which the components in the virtual layer are constructed (we discuss this further in the next section).

```pearl
1 class processor
2 3 b: bus
4 5 operations: [10] integer
6 7 read: (address: integer,
8      size: integer) -> integer
9 10 { 11    data: integer
12    data = b ! read(address, size); 13 14    reply(void);
15 16 } 17 18 write: (address: integer,
19      size: integer) -> integer
20 21 { 22    b ! write(address, size);
23    reply(void);
24 25 } 26 27 execute: (id: integer) {
28      block(operations[id]);
29 30 } 31 32 { // class main routine
33   while(true) {
34     block(any);
35   }
36 }
```

```pearl
1 class bus
2 3 mr memory
4 5 read: (address: integer,
6      size: integer) -> integer
7 8 { 9    block(Size);
10    reply(m ! read(address));
11 12 } 13 14 { // class main routine
15   while(true) { block(any) }
16 }
```

Figure 3.7: Pearl model component code samples for a simple architecture

We offer the following short description of the model behavior that is expressed by the Pearl code in Figure 3.7. All components are indefinitely waiting for calls to their methods (shown in the code listings starting from lines 25, 12 and 9 respectively). Whenever the processor component receives a read-event from the application model, a virtual processor from the mapping layer will call the processor `read()` method to model the timing consequences of the event. The `read()` method will pass (line 10) the call to the bus, which models some transfer latency (which depends on the size of data being communicated before passing the call to the memory (bus class line 8). The memory will retrieve the data located at `address`...
and the cascade of \texttt{reply} statements will return the data to the processor. Contention on the bus component is implicitly modeled: if two (or more) processor components simultaneously call methods on the bus, then the bus will first accept one of the calls and the others are stored in a message queue. The continuous wait-loop (bus class line 11) will process the queued calls one after the other, and the calling processors will incur a delay, since they made synchronous calls to the bus. Note that since the Pearl architecture model typically does not model functional behavior (just the timing consequences of the events mapped onto the architecture components), the \texttt{data} items passed between the components do not need to contain real values. To model an execute event (a virtual processor calls the \texttt{execute} method on the processor), the processor will simply block for an amount of time as was specified by the array in the YML file (Fig. 3.6 line 8). See Chapter 4 for more information on how these latency numbers are obtained.

3.3.4 The virtual/mapping modeling layer

\begin{verbatim}
<?xml version="1.0" encoding="UTF-8"?>
<mapping xmlns="http://sesame3m.sourceforge.net/YML.Map"
    side="source" name="architecture">
  <map source="A" dest="processorX" dtmpl="vproc"/>
  <map source="B" dest="processorY" dtmpl="vproc"/>
  <map source="A.out0-->B.in0" dest="memory" dtmpl="vmembus"/>
</mapping>
\end{verbatim}

Figure 3.8: Mapping YML

As was discussed before in Section 3.2.3, the virtual layer in Sesame resides between the application and architecture model layers and provides important functionalities to the model. Here we focus on the implementation aspects and show how the virtual layer can be automatically created using the template mapping mechanism. Figure 3.8 shows the YML file that specifies the (user-defined) application-to-architecture mapping for the example architecture in the previous section. The \texttt{mapping} elements construct a context where the \texttt{source} attribute refers to the application and the \texttt{dest} attribute to the architecture. Subsequently, a \texttt{map} element maps maps processes and channels to resources in the architecture model (lines 5-7). Note that the \texttt{map} element has an additional \texttt{dtmpl} attribute which specifies one of the architectural \textit{templates} associated with an architectural component.

The automatic procedure to generate the mapping layer will instantiate (for every Kahn process and channel) an object in the virtual layer corresponding to the template mapping. This is clarified in Figure 3.9 where the three Sesame modeling layers are shown. Note that every processor in the architecture model has at least one template associated with it (e.g. in architecture YML Fig. 3.6 lines 10-24). For each Kahn process, a Virtual processor (in the virtual layer) is automatically instantiated from the template as specified by the application mapping. So, although Process B and C are mapped onto the same processor P1, they ac-
3.3. IMPLEMENTATION ASPECTS

Tually will be represented by different virtual components (Q and R respectively). Similarly (although not shown in the figure), Kahn channels map onto virtual channel templates, so that the virtual layer can be fully automatically derived from the information in the mapping and architecture YML files. The template mapping mechanism makes the model more flexible, since each template may express different behavior at the virtual layer, such as for example different synchronization behavior or different model refinements (as discussed in Sections 3.2.3 and 3.2.3).

3.3.5 Graphical user-interface

A GUI has been developed in order to simplify the process of creating application and architecture YML specifications and to provide an integrated development environment (IDE) for use with Sesame. The current GUI is implemented as a plugin to the widely-used Eclipse IDE: a screenshot is shown in figure 3.10. The middle window panes show the application model (top) and the architecture model (bottom) as a graph. A model designer can change the models by selecting a tool from the palette (on the right of the graph) and drawing new nodes or connections. Pre-defined modeling components can be dragged from a library (just under the palette) and newly defined components can be stored in the library for later use. The virtual layer models and the templates from which the virtual layer is built can be edited...
CHAPTER 3. SESAME: MODELING AND SIMULATION

from a separate view (not shown in the figure). On the left side is the project manager (top) and a window to specify application-to-architecture mapping (bottom). The bottom right window shows all the properties of the currently selected node or link from the application or architecture windows. Properties can be edited and new ones can be added. The Eclipse environment can also be used to call the back-end tools to build and run the model, such as the automatic virtual layer generator, compilers and the simulator. The application shown is the MJPEG application, which we have used as an example in the previous sections. The architecture is a 4-processor crossbar based architecture: the crossbar component is shown on the right. It connects to 4 processors (note the CPU picture), by means of a bus and a local storage memory. The two components on the left of each processor are a scheduler and its scheduling policy object.

3.4 Setting up the Design Space

In the previous sections we have shown how Sesame can be used to evaluate single design points: systems consisting of an application and an architecture model with a given configuration as specified in YML. In order to make design decisions and trade-offs, designers are typically interested in comparing results between different systems and system configurations. As we have shown before, it is easy to manually modify Sesame models and their parameters – either directly in YML, or using the graphical editor. However, even a small amount of manual effort per design point is infeasible for large design spaces with thousands or millions of design points. A designer is typically interested in a specific part of the design space as delimited by some boundary parameters such as the total number of processors in the system, a finite set of alternative components, the total amount of distributed memory, etc. In order to provide a direct path from the parameter space to the objective space (as shown in Figure 1.2 of Chapter 1), we should be able to automatically generate model specifications for all design points that fall within the part of the design space delimited by the boundary parameters. In this section we will discuss two methods in Sesame to generate a set of YML specifications that represents a specific (area of) the design space. Sesame can then be iteratively run for each specification, thus evaluating all design points. We note that such techniques are not only useful for exhaustive design space search, but also for other design space search algorithms such as heuristic search algorithms (which is the topic of Chapter 7).

The design space generation approaches discussed here are quite generic and similar approaches have probably been used in the context of other modeling and simulation tools. Since each tool provides its own methods to enumerate parts of the design space, much time and effort is unnecessarily spent to create such scripts and programs. Unfortunately, standardization of even single system-level design point specifications has not yet been realized yet by the EDA community. And as far as we know, there are no standards at all for specifying a range of design points (or parts of a design space). This is unfortunate, since design space specification and generation is required in many embedded system design tools, both commercial as well as research. Lack of such standards hinders the development of new
3.4. SETTING UP THE DESIGN SPACE

Figure 3.10: Eclipse plugin for creating Sesame model specifications
methods and tools and also makes comparison of existing work more difficult. However, in recent years, things are slowly changing with e.g. the emergence IP-XACT from the Spirit Consortium. While different tools typically require a different specification format, a separation can be made between front-end and back-end specification formats: the front-end uses the design space parameters to generate abstract representations of design points; the back-end translates the abstract design point to the required format (in our case YML). This separation is recognized by IP-XACT in the sense that it is being promoted both not only as a standard to be used natively by tools, but also as an abstract interchange standard that needs to be translated to/from proprietary specification formats.

In the remainder of this thesis we will show various case studies where small and large design spaces are to be evaluated. In the following we show two design point enumeration methods that are usable with Sesame: the meta-platform approach (which we commonly use) and the generator-approach (which is still under development). A hybrid approach, which combines some of the advantages of both methods, was developed in [48].

Meta-platform approach

The first approach towards instantiating a design space is based on the idea of a meta-platform specification. In such a specification all components and parameters that could be used by any of the design points are merged into a single system specification. Individual design points are then defined uniquely by the application-to-architecture mapping specification. This is possible since in Sesame it is typically the case that architecture model components onto which nothing has been mapped (and therefore are not receiving trace events), do not influence the simulation results. Consider for example a design space which studies an architecture which has 1 to 4 processors which can be of 2 different types connected by point-to-point, crossbar or a bus network. In this case we could instantiate an architecture specification containing 8 processors (one of each type) and the 3 different interconnects. It is now relatively easy to generate automatically an exhaustive list of mapping specifications that map application processes to all possible combinations of processors and application channels to different interconnects. If, for example, the mapping specification puts all application tasks on one processor, then the remaining processor model components will simply remain inactive. If part of the exhaustively generated mappings constitute invalid design points, then they can easily be filtered out. The result is that the problem of generating the design space has been reduced to the simpler problem of generating mapping specifications and a meta-platform specification.

This approach has a number of drawbacks, the most notable of which is that it scales badly: even in small exploration case studies the number of components that have to be included in the meta-platform may be very large. Consider an example where the objective is to find optimal memory sizes within the range of 2Kb to 64Kb: with a discretization of 2Kb, a system with 4 memories would need to instantiate \((32 + 1) \times 4 = 132\) memories. The number of components multiplies further if the same component has additional properties, e.g. if there are memories with different speeds, different number of ports, etc. For real case studies where exploration may cover many types of components and properties, creating a
meta-platform becomes infeasible. Another problem is that the meta-platform specification always puts hard limits on the design space: when another DSE experiment requires a larger meta-platform, then a new meta-platform has to be designed manually. For example a meta-platform that contains 4 processors requires manual modification before it can be used in explorations with up to 8 processors. Therefore, in some cases, it is better to derive the YML system specification directly from the parameters: we call this the generator-approach.

**Generator approach**

There are many different ways to implement this approach, but here we sketch one possibility that is currently being considered for implementation with Sesame. In our case we assume that each design space parameter is linked to an operator that modifies directly the structure of the YML specification. The operator takes YML as input and produces YML as output and by putting the operators together in various ways, different design points can be generated according to the boundary parameters. An example is a processor-operator which can instantiate a processor with certain properties; more processors are added by iteratively calling the operator. Different operators add different components and properties to the architecture in such a way that a valid YML specification results after all operators have been applied. Operators may have different input requirements: for example, an operator that instantiates a bus may be able to operate on an empty input, whereas an operator that instantiates a fully connected peer-to-peer network requires the processors already to be instantiated. The order in which the YML operators are specified therefore has to occur according to a priority value associated with the operator.

```
designpoint =
emptyYML
  .addBus()
  .addProcessor(ARM)
  .changeProcFreq(200)
  .addProcessor(PPC)
  .addProcessor(MB)
  .changeProcFreq(100)
  .changeProcFreq(400, 2)
  eval( designpoint )
```

**Figure 3.11: Instantiating a single design space specification**

The advantage compared to the meta-platform approach is that the generator approach does not suffer from the same problems with respect to scaling. Furthermore, the approach is extensible: YML operators can be reused between experiments and new ones need to be defined only for any additional parameters types that need to be explored in the new experiment. Operators can be implemented as functions that work on a parsed YML tree or as programs or scripts that take YML files as input and produce YML as output. Thanks to the fact that XML is widely supported by most major programming languages (either natively or using libraries), implementing operators is not complicated and requires a fixed,
one time effort. Three operators are given a pseudo-definition in Figure 3.13. Each definition lists what requirements need to be present in the input YML, the type of operator (e.g., insertion or replacement) and the YML code to be inserted or the pattern to be substituted. For insertion operators it is specified where the inserted YML will be connected to the input YML. The example in Figure 3.11 shows how a single design point can be generated. Note the use of pseudo object-oriented function calls as operators that operate in sequence on an emptyYML object. The eval() operator is where the design point can be evaluated using (for example) the Sesame architecture simulator. Using such operators it is not hard to generate a design space according to some boundary parameters. Figure 3.12 shows a small loop that iteratively generates design space specifications consisting of a bus based system with 1 to 20 processors. Naturally, more complex combinations are possible to generate specifications for a wide variety of design spaces.

```cpp
for(num: 1 2 .. 20) {
    designpoint = emptyYML
    designpoint.addAll()
    while(num--) {
        designpoint
            .addProcessor(ARM)
            .changeProcFreq(200)
    }
    eval( designpoint )
}
```

Figure 3.12: Instantiating a range of design space specifications

### 3.5 Model and design space evaluation

As we have seen in Section 3.3, a completely defined Sesame model consists of application and architecture YML files as well as source code for the model component. It is then ready to be run as an application-architecture co-simulation, or the architecture model can be run stand-alone if the application traces have been generated and stored previously. After the model has been executed, the Pearl architecture simulator will output a large amount of statistics that have been collected during simulation time. These statistics provide system-level as well as detailed component-level information about the performance of the model. They represent an estimation of the performance of the modeled system with a certain accuracy (depending on the quality of the model). Perhaps the most frequently used statistic is that of total simulated run-time, which is given as the value of the simulated clock when the model terminates. For a model without errors or deadlocks, this is equal to the amount of clock-cycles that the modeled system needs to process all application trace-events. More detailed information includes for example the utilization of architectural components, which indicates the ratio of busy and idle times of each component. Another statistic measures system contention (indicating how many components are waiting for some other component at the same time). As Sesame uses a transparent message passing mechanism as the ba-
### 3.5. MODEL AND DESIGN SPACE EVALUATION

<table>
<thead>
<tr>
<th>operator name</th>
<th>addProcessor(type)</th>
</tr>
</thead>
<tbody>
<tr>
<td>requirements</td>
<td>primary network</td>
</tr>
<tr>
<td>operation type</td>
<td>insertion</td>
</tr>
<tr>
<td></td>
<td><code>&lt;node name=&quot;mp&quot; class=&quot;pearl_object&quot;&gt;</code></td>
</tr>
<tr>
<td></td>
<td><code>&lt;property name=&quot;class&quot; value=&quot;processor&quot;/&gt;</code></td>
</tr>
<tr>
<td></td>
<td><code>&lt;property name=&quot;type&quot; value=&quot;#type&quot;/&gt;</code></td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
<tr>
<td></td>
<td>&lt;/node&gt;</td>
</tr>
<tr>
<td>connection point</td>
<td>to primary network</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>operator name</th>
<th>addBus()</th>
</tr>
</thead>
<tbody>
<tr>
<td>requirements</td>
<td>none</td>
</tr>
<tr>
<td>operation type</td>
<td>insertion</td>
</tr>
<tr>
<td></td>
<td><code>&lt;node bus=&quot;mp&quot; class=&quot;pearl_object&quot;&gt;</code></td>
</tr>
<tr>
<td></td>
<td><code>&lt;property name=&quot;class&quot; value=&quot;bus&quot;/&gt;</code></td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
<tr>
<td></td>
<td>&lt;/node&gt;</td>
</tr>
<tr>
<td>connection point</td>
<td>none</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>operator name</th>
<th>changeProcFreq(freq, procId)</th>
</tr>
</thead>
<tbody>
<tr>
<td>requirements</td>
<td>processor procId or last added</td>
</tr>
<tr>
<td>operation type</td>
<td>replacement replacement</td>
</tr>
<tr>
<td></td>
<td><code>&lt;node bus=&quot;mp&quot; class=&quot;pearl_object&quot;&gt;</code></td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
<tr>
<td></td>
<td><code>&lt;property name=&quot;freq&quot; value=&quot;XXX&quot;/&gt;</code></td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
<tr>
<td></td>
<td>&lt;/node&gt;</td>
</tr>
<tr>
<td>replacement</td>
<td>freq</td>
</tr>
</tbody>
</table>

Figure 3.13: Definition of YML generator operators
sis of its (a)synchronous communication primitives, system contention can be measured by the number of message waiting in a component’s message queue. Other statistics include profiling information (e.g., time spent in each method of each component), bandwidth information and critical path analysis (which sequences of remote method calls block other method calls). By default, the statistics are given as a text file, which can easily overwhelm the designer. For this purpose a visualization of the statistics has been developed in the work of [99], that gives a concise summary of all the objective properties of a design point in a single view.

For real design space exploration case-studies, many design points typically need to be evaluated. We have discussed in Section 3.4 how to generate larger sets of XML specifications representing a range of system design candidates. Here we emphasize the problem of dealing with the large amount of output data that results from such experiments. In the context of the Daedalus framework, all the statistics of the Sesame simulation models will be stored in an XML-database. It also saves all input-data, so that it is possible to recreate each design point (the model complete with code and XML specification). Using the database, well-formed queries can be expressed in an XML query language which reduces the effort of writing custom scripts to analyze large volumes of simulator output statistics. In addition, we further exploit the human visual cognitive capacities in aid of design space exploration. By the same author of [97], another visualization method has been proposed which is particularly useful for evaluating large, multi-dimensional design spaces. It visualizes the design space as a large tree shape for which the leaves represent different design points. Different colors and shapes for leaves and nodes summarize important objective information about the design space (which is taken from the aforementioned XML database). Various selection and refinement mechanisms are available to the designer to re-shape the tree to interactively manipulate the visualization and to focus on design points of interest. Different metrics for visualizing (multi-objective) properties of the design points are also available (for more information we refer to [98]).

3.6 Conclusion

This chapter focused on the Sesame modeling and simulation tool which we use extensively throughout the work presented in this thesis. We have shown that Sesame uses a strict separation of concerns and a relatively high level of modeling abstraction by default. Application and architecture models are highly modular which facilitates component reuse and which further reduces the design and development time of a Sesame model. Some examples show some of the details that are involved in the creation and specification of a Sesame model. A graphical user interface has been made available to make model composition and specification even easier. We have discussed the difficulties of specifying models for multiple design points and have shown some solutions that can be used in the Sesame context and beyond. Finally we have discussed the type of information that can be obtained from the standard Sesame model.