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

Citation for published version (APA):
Chapter 5

Multi-application modeling

5.1 Introduction

In Chapter 1 we described the many ways in which modern embedded systems are becoming increasingly complex. It can be argued that this trend is driven for the most part by the increasing requirements of the end-user of the system. Apart from increasing reliability, performance and quality requirements, possibly the most challenging requirement is the integration of many functionalities on the same device. Where previously embedded systems were dedicated to a single task or a small set of tasks, modern embedded systems need to support an increasing variety of tasks. A good example is the modern mobile phone, which in addition to its primary communication function now also supports functions such as photo and video capturing, music playing, gaming, as well as browsing and office applications. The latter functions previously belonged to the domain of dedicated devices such as cameras, mp3 players, or the domain of console or desktop computing. It is a major challenge to combine all these functionalities together with additional non-functional design requirements for (mobile) embedded systems, such as power usage, cost and form factor.

So far, Sesame has only supported the mapping of a single application onto an architecture model at the time. But since modern multimedia embedded systems are increasingly multi-tasking, we need to address the modeling of effects of executing multiple applications concurrently in our system-level architecture models. In this chapter, we present two multi-application workload modeling techniques in Sesame. One technique is based on the use of synthetic application workloads while the second technique deploys only real application workloads to model concurrent execution of applications. Synthetic application workloads are particularly useful in the early design stages, since they enable (partially) parallel development of the application and architecture model. For example, the architecture model can already be tested for functional correctness while the application model is still being finalized. As will be shown in later sections, another benefit of synthetic workloads is that their parameters can be easily adapted to test the behavior of the system under specific workload conditions.

Additionally, in this chapter we will propose some ideas to combine multi-application
workloads in such a way that they contain dynamic behavior. Dynamic behavior within application models may exist at two levels: at the level of applications, or within the application at the level of processes. We refer to the former as inter-application scenarios and the latter as intra-application scenarios. Both types of scenarios need to be studied, since they can have a great impact on the workload which is to be processed by the underlying system architecture.

This chapter is organized as follows. Section 5.2 presents the two proposed multi-application workload modeling techniques for Sesame. First a synthetic multi-application model is introduced, followed by a multi-application model consisting of real applications. In Chapter 3 we discuss how various features in Sesame help the designer to create multi-application models with relatively little effort. Subsequent Sections 5.4.1 and 5.4.2 deal with the modeling and representation of dynamic inter and intra-application workloads in Sesame. Experiments showing the various techniques presented throughout the chapter are in Section 5.5. Related work is in Section 5.6 and we conclude the chapter in Section 5.7.

5.2 Multi-application workload modeling

As mentioned before, Sesame has up to now only supported the mapping of a single application onto an architecture model at the time. Modern multimedia embedded systems are however increasingly multi-tasking. Therefore, we need to address the modeling of effects of executing multiple applications concurrently in our system-level architecture models. To this end, we propose two multi-application workload modeling techniques. One technique, which we will discuss first, is based on the use of synthetic application workloads while the second technique deploys only real application workloads to model concurrent execution of applications.

5.2.1 Synthetic multi-application workload modeling

Multi-application modeling using synthetic application workloads is illustrated in Figure 5.1. Note that the FIFO buffers between virtual processors are not depicted in Figure 5.1 for the sake of simplicity. On the left-hand side, a Sesame system-level model with a single primary application is shown. The three processes in this application are mapped onto two processing cores (P0 and P1) in the underlying architecture. Since processes A and B are mapped onto the same resource, a scheduler named Local-Scheduler (or L-Scheduler) is used for scheduling the workloads (i.e., application events) from both processes. However, a second level of scheduling hierarchy is added by introducing so-called Global-Schedulers (or G-Schedulers). These global schedulers are basically equivalent to local schedulers in terms of functionality but instead of intra-application events they schedule application events from different applications. Evidently, the local and global schedulers can also deploy different scheduling policies. When, for example, the interleaving of processes inside an application is statically determined at compile time, the local scheduler can model this by ‘merging’ the events from the event traces according to this given static schedule. At the same time, the
global scheduler can schedule application events from different applications in a dynamic fashion based on, for example, time slices, priorities, or a combination of these two. Here, we would like to note that although the schedulers support preemptive scheduling, this can only be done at the granularity of application events. The simulation of a single application event is atomic and thus cannot be preempted in Sesame. Furthermore, we currently do not model any overheads caused by the context switching itself (e.g., OS overhead, cache misses, etc.). This is considered as future work.

In synthetic multi-application modeling, the application events external to the primary application (see Figure 5.1) are generated by a stochastic event generator. Hence, this event generator mimics the concurrent execution of one or more application(s) besides the primary application. Based on a stochastic application description, which will be discussed later on, the application generator generates traces of Ex(ecute), Read and Write application events and issues these event traces to special virtual processors, indicated by VP in Figure 5.1. Multiple instances of these event generators, each with their own stochastic application description, can be used to model concurrent execution of more than two applications.

The virtual processors (VP) used for the trace events from the stochastic event generator are special in the sense that they, unlike normal virtual processors, are not connected to each other according to the application topology (see Section 3.2.3 in Chapter 3). Rather than explicitly modeling communication synchronizations, a VP models synchronization behavior stochastically. To illustrate the interactions between the event generator, a VP and a global scheduler of a system-level model, consider Figure 5.2. The figure shows these interactions in the case of an "Ex(A), Ex(B), Read, Write" event sequence is generated by the event generator. At (simulation) time t₀, the Ex(A) event is consumed by the VP. The VP immediately forwards this event to the global scheduler it is connected to, and waits for an acknowledgment from the scheduler. After the Ex(A) event has been scheduled for execution on the architectural resource (taking T(sched) time units) and the actual execution (taking T(A) time units), control is returned to the VP by sending it an acknowledgment. Hereafter, the VP can consume another application event again. In the case of the example in Figure 5.2, the VP now consumes the Ex(B) event which is handled in an identical fashion as the Ex(A) event. However, VP handles the Read and Write events, which are consumed at times t₂ and t₃ respectively, in a slightly different way. Instead of directly forwarding these events to the global scheduler, like is done with Ex events, VP now first models a synchronization latency. This latency refers to the time the read and write transactions need to wait for data or room in the

---

**Figure 5.1: Multi-application modeling using synthetic application workloads.**
buffer from/to which is read/written. The synchronization latency, indicated by \( T(\text{sync}) \) in Figure 5.2, is a stochastic parameter of \( \text{VP}_S \), as discussed below.

![Figure 5.2: Interaction between Virtual Processor (VP) and Global-Scheduler in synthetic multi-application modeling.](image)

Table 5.1: Parameters for the synthetic application workload generation.

<table>
<thead>
<tr>
<th>Stochastic event generator parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( A_{E_x} )</td>
<td>Set of possible Ex(ecute) application events</td>
</tr>
<tr>
<td>( P_{E_{x,t}} ), with ( \sum_{i \in A_{E_x}} P_{E_{x,t}} = 1 )</td>
<td>Probabilities of the different events in ( A_{E_x} )</td>
</tr>
<tr>
<td>( r_{\text{comp}} \cdot r_{\text{comm}} )</td>
<td>Computation to communication ratio</td>
</tr>
<tr>
<td>( r_{\text{read}} \cdot r_{\text{write}} )</td>
<td>Read to write ratio</td>
</tr>
<tr>
<td>( M ) with ( \sum_{i \in M} P_{M_i} = 1 )</td>
<td>Set of possible message sizes</td>
</tr>
<tr>
<td>( P_{M, t} ), with ( \sum_{i \in M} P_{M, t} = 1 )</td>
<td>Probabilities of the different message sizes</td>
</tr>
<tr>
<td>( \text{NP} )</td>
<td>Number of communication ports</td>
</tr>
<tr>
<td>( P_{\text{port}, t} ), with ( \sum_{j=0}^{\text{NP}} P_{\text{port}, t} = 1 )</td>
<td>Probabilities of the different port usages</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>VP ( S ) parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{Sync}_{\text{Read}} )</td>
<td>Mean synchronization latency for reads</td>
</tr>
<tr>
<td>( \sigma_{\text{Read}} )</td>
<td>Standard deviation of read latencies</td>
</tr>
<tr>
<td>( \text{Sync}_{\text{Write}} )</td>
<td>Mean synchronization latency for writes</td>
</tr>
<tr>
<td>( \sigma_{\text{Write}} )</td>
<td>Standard deviation of write latencies</td>
</tr>
</tbody>
</table>

Table 5.1 lists the parameters used by the stochastic event generator as well as a \( \text{VP}_S \). These parameters can be specified both globally – describing the behavior for all traces (for the event generator) or ports (for a \( \text{VP}_S \)) – and on a per-trace/per-port basis. Descriptions on a per-trace/per-port basis overrule global descriptions, in the case there is an overlap of both types of descriptions. The parameter \( A_{E_x} \) specifies the set of possible Ex events that can be generated. For example, \( A_{E_x} = \{ \text{DCT, VLE} \} \) specifies that Ex(DCT) and Ex(VLE) events can be generated. \( P_{E_{x,t}} \) describe the probabilities of the events in \( A_{E_x} \).
The ratio’s \( r_{\text{comp}}:r_{\text{comm}} \) and \( r_{\text{read}}:r_{\text{write}} \) specify the computation to communication ratio and read to write ratio, respectively. So, for example, by increasing the \( r_{\text{comp}}:r_{\text{comm}} \) ratio, the application behavior can be made more computationally or communication intensive. The parameter \( M \) specifies the set of possible message sizes that can be used in communications. In multimedia applications, application data is often communicated in fixed data chunks (e.g., pixel blocks) from one application phase to the other. \( P_M \) specify the probabilities of the different message sizes. \( N_P \) denotes the number of communication ports for which read and write transactions can be generated. \( P_{\text{port}} \) are the probabilities of the different port usages. Again, all of the above parameters can be specified globally (valid for all event traces) or on a per-trace basis.

The VP\( S \) parameters \( \text{Sync}_{\text{Read}} \) and \( \text{Sync}_{\text{Write}} \) specify the mean synchronization latency for read and write transactions, respectively. \( \sigma_{\text{Read}} \) and \( \sigma_{\text{Write}} \) contain the standard deviations of the two aforementioned means. By default, a VP\( S \) uses an Erlang distribution to determine synchronization latencies. These VP\( S \) parameters can again be specified globally (valid for all communication ports of a VP\( S \)) or on a per-port basis.

Figure 5.3: Multi-application modeling using realistic application workloads.

### 5.2.2 Realistic multi-application workload modeling

In our second multi-application workload modeling technique, we realistically model the concurrent execution of multiple applications. That is, multiple Kahn application models are actually executed concurrently, as shown in Figure 5.3, and produce realistic event traces that are again scheduled on the underlying architectural resources using the global schedulers. In contrast to synthetic workload modeling, the secondary KPNs use normal virtual processors in the mapping layer. Hence, synchronization behavior in the parallel applications is modeled explicitly for all participating KPN applications (i.e., there is no difference between primary and secondary applications). This implies that, when considering Figure 5.2, the \( T(\text{sync}) \) now refers to the actual synchronization times between application processes.
Moreover, the secondary KPNs also require L-schedulers to merge (i.e., schedule) event traces when multiple application tasks are mapped onto a single architecture resource. Naturally, the policies of the L-schedulers can vary between the different KPN applications taking part in the system simulation. When considering Figure 5.2, we now have $T(sched) = T(L\text{-sched}) + T(G\text{-sched})$ for all participating KPNs.

5.3 Multi-application modeling: a designer’s perspective

In the previous sections (as well as the remainder of this chapter) different modeling techniques are discussed that extend Sesame’s modeling capabilities. In this section we discuss how these modeling extensions can be implemented in Sesame in such a way that the implementation effort is minimal. As we are targeting models for use in the early stages of system design, it is important that all techniques can be used in an easy and straightforward way, in order not to slow down the design process. In this section we will describe some of the ways our tools can support reduction of the modeling effort for the previously proposed techniques.

In Chapter 3 the different layers within a Sesame model and their respective role in the system model were shown in detail. It was described in Section 3.2.3 that the virtual layer is automatically instantiated before the start of the simulation. This instantiation process uses templates which are associated with each processing or communication component in the architecture model. The application mapping (which maps tasks and channels to their respective targets: processing and communication resources in the architecture layer) has an additional parameter to specify which template to use for each component in the virtual layer. In this way, different synchronization behaviors can be contained in different templates and the virtual processors will be automatically instantiated with the right synchronization behavior. Since they implement a special kind of alternative synchronization behavior, stochastic virtual processors (i.e. $VP_3$ in Figure 5.1) are implemented as templates too. In Figure 5.1, the details of the stochastic event generator are not shown, but it does in fact consist of multiple (stochastic) processes (just like a regular KPN). Therefore, mapping stochastic processes is as easy as mapping ordinary processes, with the only addition that the $VP_3$ parameters need to be specified (see Table 5.1). Moreover, when the designer uses the model specification GUI (Section 3.3.5), the designer can simply drag-and-drop a generic processor component from the library, which already includes a $VP_3$ template.

Secondly, we discuss the process of creating a stochastic application model. In Figure 5.1, the stochastic application event generator is shown without details such as individual stochastic processes and their channels. Creating the internal topology of the stochastic application event generator is currently a manual process: using the GUI, the designer can drag stochastic processes onto the canvas and connect them with channels in any topological structure. Again, the designer then only needs to fill in specific values for the latency tables which can be done either globally or on per-port/per-trace basis (Table 5.1). There can be situations where the designer wants to quickly instantiate a KPN without detailed control over the topology. For this purpose we envision a new tool for the GUI that instantiates a
5.4. DYNAMIC APPLICATION BEHAVIOR

Topology according to a few parameters, such as the number of synthetic processes and the average number of channels per process. Using only a few parameters, various (a)cyclic, low or fully connected KPN topologies can be instantiated which the designer can further tailor to need. Instantiation of stochastic virtual processors is again done automatically, using the template mapping as mentioned before.

The final tool-related issue that we address here concerns the parameters for the stochastic virtual processors (Table 5.1). In particular we propose a way to automatically derive initial values for the VP$_3$ parameters. As observed before in Section 3.2.3, the synchronization latencies of virtual processors consist of all kinds of delays. E.g., when a READ event is blocked this may be due to the fact that the writing process is waiting for some processes that are occupying shared resources (e.g., processors or memories). A possible initial setting for the VP$_3$ parameters is to base the mean synchronization latency on the workloads of all other processes that share the same resource. For example, we could use the average of all EX event latencies for processes that share the same resource. As an alternative to the stochastic virtual processor, we propose an auto-tuning stochastic virtual processor that “learns” an acceptable set of synchronization latencies during a trial-run simulation. Initially, for each channel connected to the virtual processor an initial synchronization latency of 0 is used. This value is automatically adjusted at model runtime by counting the workload and frequency of EX events that share the same resources. For example, the $\text{Sync}_{\text{Read}}$ value of a process is auto-tuned to the average workload of the process that writes to that channel. Note that there are various sources of inaccuracy in the auto-tuning method. For example, there may be other latencies in the architecture model (other than the E$\text{x}$ events) that contribute to the synchronization latency. Despite such inaccuracies, the auto-tuning synthetic virtual processors provide initial values for the VP$_3$ parameters, which can be refined by the designer.

We conclude that the designer has a range of options available to create a stochastic application model in the early stages of design with a relatively small engineering effort. Additional effort is necessary only in cases where if the designer wants more detailed control of the stochastic application’s properties or when the designer wants the stochastic application to match the behavior of a primary (real) application model. An automatic topology generator and auto-tuning synthetic virtual processors were introduced as future Sesame extensions to reduce even further the effort to create synthetic models.

5.4 Dynamic application behavior

In this section we discuss how Sesame’s real and synthetic application models can be adapted to represent dynamic application behavior. We distinguish two types of dynamic application behavior: inter-application behavior (between applications) and intra-application behavior (between processes within a single application KPN). A variation of the dynamic inter-application behavior presented in the first subsection below, has been used in a case study that considers a partially dynamic reconfigurable architecture (see Section 6.5). Dynamic intra-application behavior is already available in some of the realistic application models.
that we use in Sesame. However, the synthetic application generator from Section 5.2 does not explicitly support this. In Section 5.4.2, we propose a method to add dynamic intra-application behavior to stochastic Sesame models.

### 5.4.1 Dynamic inter-application behavior

A multi-application model in Sesame consists of two or more disjunct KPNs, which have no dependencies other than being mapped onto shared resources. Although this is a perfectly valid Sesame application model, it is unable to capture important dynamic inter-application behavior that is particularly relevant to modern embedded systems where tasks may enter and leave the system at any given time. For example: when a mobile-phone user takes a picture, the camera application tasks will put a temporary additional load on the system’s resources. In this way, application loads can be generated with a huge dynamic range: an additional application can for example double the workload on the underlying architecture. These types of workloads are sometimes called user-scenarios, since the arrival of a new application is often (but not necessarily) initiated by the user or the environment. We will refer to them as inter-application scenarios to distinguish them from intra-application scenarios (the topic of the next section). In the following, we will shortly discuss two methods that allow the modeling of dynamic inter-application workloads in Sesame. Some of the problems that need to be solved are 1) KPNs are not naturally suited for modeling dynamic/reactive behavior at the inter-application level and 2) KPN is an untimed model of computation which complicates the specification of the arrival *time* of a sporadic application.

![Diagram](image)

Figure 5.4: Example of a multi-application workload with dynamic inter-application behavior using a Markov-model orchestrator node.

Here we will assume that each KPN in our multi-application model has one or more "source" nodes as this is common in our targeted streaming media application domain.
5.4. DYNAMIC APPLICATION BEHAVIOR

Source nodes have no initial dependencies on other nodes and are therefore immediately runnable. An example source node is a node that feeds raw input data into the application (either from a framebuffer memory, a file, or a raw data stream: camera, or microphone, etc.). Therefore, the KPN is usually alive for as long as the source node feeds data into the KPN.

Now, suppose we create a multi-application model in Sesame consisting of two or more KPNs with source nodes as described above. Sesame’s application simulator will start running all KPNs concurrently and therefore, by default, we can only model the single scenario where all applications start running at the same time (though possibly with different priorities as specified in the G-schedulers in the virtual layer. In the first method to model dynamic multi-application workloads, we remedy this situation by introducing an orchestrator node as shown in Figure 5.4. The purpose of the orchestrator node is to put an artificial (read) dependency on each source node such that each of the KPNs can only start according to a scenario defined by the orchestrator. Note that in this way the orchestrator can not only introduce applications to the scenario, but also remove them. For example, let us assume that the second application in Figure 5.4 is a video encoder and that source node S2 (after the orchestrator dependency READ (Orchestrator) reads out a framebuffer (represented by the EX(function) event) and passes it as macroblocks to node A. The orchestrator can include the application in the current scenario by sending a token on its corresponding output control channel. Vice versa, it can exclude the application by not sending the token. Using the control tokens, the orchestrator can create those scenarios and scenario transitions that are of interest to the designer. We propose here to use a Markov chain to represent the transitions between scenarios where each state represents a scenario (each with its respective control tokens) and transitions are given by probabilities in the normal way (see Figure 5.4).

One problem with the orchestrator approach described above, is that it is impossible to define transitions between user-scenarios at certain time intervals. This would be necessary to implement the behavior where the system reacts to externally timed events such as the occurrence of interrupts (e.g., a user presses a button on the TV’s remote control after which teletext is started as a picture-in-picture application on the screen). Since KPNs are an untimed model of computation, it is not possible to express the behavior "wait n time units and then start application X". For this purpose, we define a special ‘SLEEP(N)’ application event, which basically indicates that an application process is not active during a period of N time units. The SLEEP event is created by a special event annotation in a Kahn process and it takes n as its only argument. As with all events, it is passed to the virtual layer, where it is consumed by a virtual processor (it is not passed on later to the architecture model). The virtual processor exists in the same timed simulation domain as the architecture model, so that we can implement the desired behavior. In the virtual processor, the SLEEP event causes the virtual processor to sleep (i.e. block in virtual time) for the specified period. While the virtual processor is blocked in this way, it will be unable to continue its normal processing of EX, READ, and WRITE events, thus effectively suspending the application process. Note that by suspending one process in the KPN of an application, the other processes will soon stall because of (in)direct unmet dependencies on the suspended process. We can prevent
suspending the application mid-stream (with unprocessed data before or after the suspended process) by issuing sleep events only to the source nodes (this situation is shown in Figure 5.5).

![Figure 5.5: Example of a multi-application workload with dynamic inter-application scenarios implemented using the SLEEP application event.](image)

Evidently, the SLEEP events provide the opportunity to freeze the issuing of application events for a while, which basically mimics sporadic or periodic execution behavior of applications. The sleep event technique will work transparently for both realistic and stochastic application workloads alike. Compared to the sleep-event technique, the orchestrator technique has the advantage that the scenarios and use-cases can be defined in a single location in the model. These two techniques enable the designer to assess a variety of different scenarios or use cases [36]. A technique based on the orchestrator approach will be used in Chapter 6 to create a pseudo-dynamic workload for an architecture with dynamic reconfiguration capabilities.

### 5.4.2 Dynamic intra-application behavior

The stochastic application workload proposed in Section 5.2 consists of stochastic processes (and stochastic virtual processors) which use a parameterizable, yet fixed event generation distribution. In realistic non-trivial applications, it is often the case that the application (and its separate processes) move through different execution phases. We would like the synthetic trace event generator to create events according to a unique probability distribution in each of those phases. We refer to each possible combination of simultaneously occurring process phases as an intra-application scenario. They differ from user scenarios (or inter-application scenarios) as they occur at the process (task) level within applications and are typically not directly related to any action by the user.
In Sesame, there are multiple ways in which intra-application scenarios can be specified for the purpose of stochastic modeling. We propose here a solution analogous to the previously discussed orchestrator process by using a Markov chain to traverse different process phases. In this case, each stochastic process contains multiple probability tables: one for each scenario in which it can be involved. Note that if a stochastic process is not part of an explicit scenario, then it simply uses its default probability table. The Markov model exists outside of the application graph and each of its states represents a scenario. The node containing the Markov model is connected by control channels to all stochastic processes that contain multiple probability tables. The granularity with which the Markov model changes state is to be determined by the designer. For example, if a node is a typical streaming process containing a Read-Execute-Write loop, then it makes sense to advance the Markov model at the beginning of the loop.

In Figure 5.6, an example is given where the application can run in two distinct scenarios (1 and 2). Processes A and C can take part both in scenario 1 and 2; depending on the value from their respective control channel they use either probability table 1 or 2. The Markov model lists a high probability for scenario 2, but approximately 10% of the time, the application jumps back to scenario 1. Processes S and B have the same behavior in each scenario and therefore contain only a single (default) table. In order to synchronize the switching between scenarios, it may be necessary that the Markov model (IMM) is implemented outside the KPN, not adhering to normal KPN communication rules. Note that an implementation of the proposed intra-application method has not yet been implemented and remains as future work.
Figure 5.7: Estimated execution times of concurrent execution of M-JPEG and producer-consumer applications. The latter is parameterized in both computation and communication grain size.

5.5 A preliminary case study

For illustrative purposes, we performed a small experiment using the multi-application workload modeling support now available in Sesame. More specifically, we modeled two Kahn applications that execute concurrently. The first (and primary) application is a Motion-JPEG (M-JPEG) encoder, and the other one is a synthetic ‘producer-consumer’ application transferring data from producer to consumer. The M-JPEG application encodes 8 consecutive 128x128 resolution frames, while the producer-consumer application is parameterizable in both computational and communication load. That is, the producer iteratively models a parameterizable computing latency after which it sends a parameterizable chunk of data to the consumer. In our system-level model, both applications are mapped onto a multi-processor SoC, containing 4 processors with distributed memory and connected through a crossbar switch. We applied a simple round-robin policy for scheduling tasks from both applications at the G-schedulers (see Section 5.2.2).

Figure 5.7 shows the estimated system-level execution times (combined for both applications) when varying the computation and communication grain sizes of the producer-consumer application. As can be seen from Figure 5.7, the results show a quite predictable behavior, which helps to gain trust in our multi-application modeling method. That is, the system performance is only marginally affected for small computation and communication grains of the producer-consumer application. But after a certain threshold, the producer-consumer application starts to dominate the system performance (computation-wise, communication-wise, or both).

As future work, we plan to actually validate multi-application modeling results against a Daedalus prototype implementation. Note that for a useful validation, we may have to extend the model to capture latencies associated with the (embedded) operating system that
schedules the tasks from multiple applications. This would be necessary, for example, if there is a significant context switching overhead. We also note that the L-schedulers have been frequently used in single-application workloads to enforce a specific scheduling of application tasks. For example, in the validation case studies in Chapter 8, the modeled MPSoC uses a static scheduling of tasks. Therefore, a scheduler is associated with each processor in order to schedule events according to a fixed, static scheduling policy.

5.6 Related work

The modeling of (parallel) workloads for the purpose of performance analysis is a well-established research domain, both in terms of realistic and synthetic workload modeling (see e.g. [56, 28, 91]). A recent focus area is, for example, statistical simulation for micro-architectural evaluation [23]. In this technique, a stochastic program description, which is a collection of distributions of important program characteristics derived from execution profiles, is used to generate synthetic instruction traces. These synthetic traces are subsequently used in trace-driven processor and/or memory-hierarchy simulations. This work focuses on generation of applications at the instruction level, since it is targeted towards execution on instruction set simulators. Although we target a different domain (system-level multi-processor models), it is interesting to compare the similarities and differences between the approaches. The work of [22] is motivated by the need to find workload representations (a stochastic model) that have smaller storage requirements than instruction traces from actual applications (since reading large traces slows down simulation time significantly). On the other hand, our high-level event traces for a single application are fairly small and trace sizes are only problematic in very special cases (see the work of [111] described below). Additionally, compared to [22] our work is not focused on finding stochastic models that accurately represent workloads of actual applications. Instead, our focus is on providing the designer with tools to generate stochastic workloads for use in the very early design stages when the final workload of the system may still be (partially) unknown. Where [22] produces traces with syntactically correct read and write dependencies, the stochastic event generators presented in this chapter do not. Instead, we include the read/write dependencies in the stochastic model itself to give the designer maximal flexibility to experiment with different workload characteristics.

Another area in which synthetic workload modeling has recently received a lot of attention is network workload modeling for network-on-chip simulations [113, 103, 65]. In [62, 68], multimedia application workloads are described and characterized analytically using so-called variability characterization curves (VCCs) for system-level performance analysis of multi-processor systems-on-chip. These VCCs allow for capturing the high degree of variability in execution requirements that is often present in multimedia applications. A fair number of research efforts addressed the high-level modeling of a RTOS to be used in system-level models for early design space exploration [35, 41, 60]. Rather than focusing on how to model multi-application workloads, these efforts mainly address abstract modeling of RTOS functionality, efficient simulation of this functionality, and refinement of these
abstract RTOS models towards the implementation level.

In the work of [77] design space exploration case studies using a modified simulated annealing technique are performed on randomly generated KPN graphs. A closer look at their KPN generator reveals that it takes only a few simple parameters to create KPNs with random topology, communication patterns and execute annotations. These parameters specify global properties of the KPN: the number of processes, number of communication events, size of communicated data, execution time and avg. number of channels per process, etc. In contrast to our stochastic event generator, their approach produces syntactically correct event traces, but there is very little control on the properties of specific processes and channels. In this way, some properties that are typical streaming media applications are not captured, e.g. the fact that communication or execution events for a specific process often have a limited choice respectively for data size or event type. Of course, there may be situations where the designer does not require those workload properties. Therefore the KPN generator from [77] is a useful addition to the Sesame toolbox (in fact, we use it in Chapter 8).

The idea to represent transitions between different scenarios or application process phases as a Markov model is not new (see for example [102]). Markov chains provide a clear and natural way to model transitions between a finite number of states with (simulated) non-deterministic behavior. In our approach the Markov model determines the stochastic event generator’s probability functions: each state corresponds to a different set of probability functions for certain processes in the KPN. In [102], the Markov model is integrated in an extended data-flow (SDF) model called Scenario Aware Dataflow (SADF). SADF integrates special control nodes (detectors) into the application graph, which use control channels to steer the current scenario to each of the nodes that participate in a certain scenario. In each firing, the detector determines the new scenario according to one of its integrated Markov models. This is quite similar to our approach, except that in our case the control channels "select" the process’ probability table, whereas in their approach it selects a specific token production and consumption rate. Another difference is that in SADF the state advancement of the Markov model occurs with every firing of the detector node, whereas in our case it is user-defined. We note that the SADF models in are more suitable for static analysis, but are not as expressive as KPN models. Another difference is that the SADF work in [102] has a focus on approximating realistic application workloads, which in our case is not the primary concern. For an extensive review of application modeling approaches based on the integration of finite state machines with dataflow models, we refer to [26].

A very interesting continuation of multi-application modeling performed in the context of the Sesame simulation environment is in [112, 110]. From this work we borrowed the intra and inter-application scenario terminology used throughout this chapter. The main contribution of [111] consists of a method to efficiently identify intra and inter-application scenarios from a multi-application workload as well as a co-exploration technique based on genetic algorithms. Scenarios for a certain workload are identified by executing a multi-application model (may contain both real and stochastic applications) and analyzing the generated event traces. The scenarios are then stored in a tree-structured database where the leaf nodes consist of all unique trace segments that can be produced by a single process. One level higher
in the tree stores a representation of the intra-application scenarios: it lists (for each application) combinations of communicating trace segments of individual processes that occur in the workload. At the root of the tree the inter-application scenarios indicate which of the applications in the workload were identified to execute simultaneously. By efficient loop-compression of the trace segments in the leaf nodes, this hierarchical scenario database has minimum storage requirements and is subsequently used for the co-exploration to identify solutions (architectural design points/mappings) for the problem space (all possible scenarios as stored in the database). The problem to be solved is that evaluating all possible scenarios for all possible design points is infeasible and that it is very hard to find a smaller subset of scenarios that represents all scenarios. The proposed co-evolution technique is shown to produce good results by simultaneously searching for optimal design points and a representative subset of scenarios.

Note that the work in [111] and the work in this chapter are complementary, but are geared towards different phases in the design process. Where [111] has a focus on full-blown design space exploration (when the system’s workload is known), the work in this chapter is geared towards early model development (when the final workload is typically unknown). Moreover, the work in [111] does not consider the dynamic inter-application phase transitions other than those that were present during scenario identification. In order to create new inter-application scenarios, the orchestrator-technique from this chapter could be applied.

5.7 Conclusion

In this chapter we have shown the mechanisms and techniques provided by Sesame to model multi-application workloads. This is required, since multi-application workloads are increasingly becoming standard in modern embedded system design. Furthermore, we have shown that Sesame has basic support for modeling synthetic in addition to the realistic application workloads. Initial support is available to model application workloads with dynamic inter- and intra-application behavior. The case study illustrated that stochastic modeling using synthetic trace generators is a useful tool to answer various "What if..?" questions about a system design candidate. For example: what happens to primary application performance if (after a certain time) an additional task with certain properties enters the system? Furthermore, an important benefit of stochastic application models is that they enable simultaneous co-development of application and architecture models. For example, the architecture model can be tested with one or more stochastic application models while the application model is still under development. This is a realistic case, since significant effort is involved in making an application model suitable for mapping onto an MPSoC by transforming it (for example) in a KPN. Such a co-development of application and architecture model can result in a significant reduction in the early stages of system design. In order not to increase development time at these early design stages, we consider it important that the techniques introduced in this chapter do not penalize the designer with a lot of additional implementation time. Therefore, we put quite some emphasis on the user-friendliness of the various stochastic
multi-application modeling techniques. We mentioned the possibility to use the library to contain standard scheduling components, scheduling policies and standard stochastic application generator components. Furthermore, the template mapping strategy proved to be helpful to reduce the effort that is otherwise associated with creating a stochastic application model. However, we have not yet deployed the techniques from this chapter (in particular the stochastic application models) in a large scale case study. This remains as future work.