DSPE: DOMAIN-SPECIFIC LANGUAGE FOR PARALLEL REAL-TIME STREAM PROCESSING

A Model-Driven Generative-Programming Approach for Event-Driven Scheduling and Dynamic Load Balancing on Multi-Core Processors and Graphical Processing Units (GPUs)
# Contents

List of Figures vii  
List of Tables ix  
List of Abbreviations xi  

## 1 Application Family: Real-Time Stream Processing 1  
1.1 Main Characteristics of Stream-Processing Applications 2  
1.2 Common Features of the Application Design and Architecture 3  
1.3 Models of Computation and Parallel Models 5  

## 2 Domain-Specific Development Techniques 9  
2.1 Rapid Prototyping and Software Customization 9  
2.2 Model-Driven Generative Programming and Complexity 11  
2.3 Component-Based and Pattern-Based Code Generators 12  
2.4 Aspect-Oriented Domain Modelling 13  
2.5 Domain-Specific Techniques and Parallel Programming 13  
2.6 Domain-Specific Languages 14  
2.7 Extension Support in the DSPE Model and Generators 15  
2.8 Development Challenges of Domain-Specific Languages 17  

## 3 DSPE Base Software Architecture 19  
3.1 DSPE Functional Elements 19  
3.2 Programming Languages Used for the Generated Code 21  
3.3 Design Patterns Used in the Base Software Architecture 21  
3.4 Object-Oriented Development in C 22  
3.5 File Organization of the Generated Code 24  
3.6 Class Hierarchy of the DSPE Objects 25  
3.7 DSPE Components 27  
3.8 Software Units 27  
3.9 Gates 28  
3.10 Parameter vs. Data Gates 29  
3.11 Structures 29  
3.12 Configurations 30
## Contents

3.13 Composites .................................................. 30
3.14 Runners .................................................. 31
3.15 Application Execution: Lifecycle Phases .................. 33
3.16 Unit Behaviours ............................................. 34
3.17 Runner Behaviours .......................................... 35
3.18 Wrap Unit Behaviours ......................................... 35
3.19 Block Processing .............................................. 35
3.20 Managed Pointer Gates and In-Place Processing .......... 37
3.21 Composites and Block Processing ......................... 38
3.22 The Critical Section .......................................... 40
3.23 Managers .................................................. 43
3.24 Memory and Thread Management Infrastructures .......... 44
3.25 Errors, Console/Log Information, and Execution Profiles . 44

4 Parallel Stream-Processing Approaches ......................... 45
4.1 Parallelism in Stream-Processing Applications: Granularity 46
4.2 Parallelism in Stream-Processing Applications: Decompositions 48
4.3 Parallel Design Patterns for Stream Processing ............ 50
4.4 Performance Improvements from Parallel Stream Processing 51
4.5 Data Dependencies and Bottleneck Components ............. 51
4.6 Technical Aspects of Shared-Memory Multiprocessors .... 53
4.7 Technical Aspects of Preemptive Operating Systems ...... 54
4.8 Stream Decomposition: Grain Size ........................ 55
4.9 Run-Time Interferences, Execution Pressure, and Load Imbalances 56
4.10 Non-Blocking Synchronisation: Spinlocks and Atomic Actions 59
4.11 Lightweight Scheduling and Synchronization Infrastructure 60
4.12 Scheduling Approaches for Parallel Stream Processing .... 61

5 Event-Driven Infrastructure ..................................... 63
5.1 Discrete Events and Message Passing ....................... 64
5.2 Abstract Types of the Event-Driven Infrastructure .......... 65
5.3 Events and Event Gates ........................................ 66
5.4 Memory Management: Event Pools .......................... 68
5.5 Event Units and Queue Units .................................. 69
5.6 Support for Events in Unit Behaviours ....................... 71
5.7 Advanced Event Support: Transit-Discard and Arm-Post .... 72
5.8 Schedulers .................................................. 73
5.9 State Unit Behaviours ......................................... 76
5.10 Automatic Support for Events ................................. 76
5.11 Special-Purpose Lifecycle Phases for Events ............... 77
5.12 Events and Composites ....................................... 79
5.13 Events and Runners ........................................... 80
5.14 Mixing Configurations and Schedulers ....................... 81
## 6 Soft Coprocessor Infrastructure

6.1 Abstract Types of the Soft Coprocessor Infrastructure ............................................ 84
6.2 Lightweight tasks, Coprocessor Units, and Soft Coprocessors ................................ 85
6.3 Design Patterns Used in the Soft Coprocessor Infrastructure ................................... 86
6.4 Integration of Soft Coprocessors with the Event-Driven Infrastructure ....................... 87
6.5 Memory Management: Task Pools ................................................................. 88
6.6 Task Immutability and Restricted Access Time .................................................. 89
6.7 Coprocessor Schedulers ................................................................................. 90
6.8 Coprocessor Unit Behaviours ......................................................................... 92
6.9 State-Less vs. State-Full Algorithm Support .................................................. 94
6.10 Automatic Support for Tasks ....................................................................... 96
6.11 Coprocessor Manager .................................................................................. 97
6.12 Work Sharing and Work Stealing ................................................................. 100
6.13 Long-Running and Blocking Tasks ............................................................. 103
6.14 Application Execution Pressure .................................................................... 104
6.15 Scheduler Contribution and Scheduler as a Task ......................................... 105
6.16 Pull and Push Execution Pressure .................................................................. 106
6.17 Execution Pressure Patterns ....................................................................... 110
6.18 Event-Driven Delay Line Support .............................................................. 111
6.19 Thread Management Done by the Coprocessor Manager .............................. 112

## 7 Accelerator Infrastructure

7.1 Stream Processing with Multi-Cores and GPUs ................................................ 116
7.2 Design Guidelines for Stream Processing with Heterogeneous Processors ............ 118
7.3 Integration of the Accelerator and the Soft Coprocessor Infrastructure ............... 119
7.4 GP Unit Behaviours ..................................................................................... 120
7.5 Automatic Support for Kernels ...................................................................... 121
7.6 Special-Purpose Support for Kernel Tasks ................................................... 122
7.7 Integration of CUDA Streams ...................................................................... 123
7.8 Memory Management: GP States and GP State Pools ..................................... 124
7.9 Heterogeneous Work Stealing ..................................................................... 125
7.10 Support for More Than One Accelerator ..................................................... 126
7.11 GPUs and the Application Execution Pressure ............................................. 126
7.12 Performance Considerations About GPU Kernels ......................................... 127

## 8 DSPE Model

8.1 EMF .......................................................................................................... 130
8.2 Base Model ................................................................................................ 131
8.3 DSPE Model Entities .................................................................................. 135
8.4 DSPE Sub-Models ..................................................................................... 135
8.5 DSPE Model Definitions ........................................................................... 136
8.6 DSPE Model Aspects ............................................................................... 136
8.7 Software Unit Definitions and Composite Definitions .................................. 138
List of Figures

1.1 Example of networked digital filters ............................................ 1
1.2 Use of buffers for stream processing ........................................ 3
1.3 Macrostructure of a stream-processing application ....................... 4
1.4 Example of a graph composed of vertices and arcs ....................... 5
2.1 Development process used with DSPE ........................................ 11
2.2 Abstraction levels of programming languages ............................. 14
3.1 DSPE functional elements ....................................................... 20
3.2 DSPE elements of the base software architecture ....................... 25
3.3 Functionality of Gates in Composites ....................................... 31
3.4 Statechart diagram of Runners .............................................. 32
3.5 Application lifecycle phases .................................................. 34
3.6 Schematic structure of an application without block processing ....... 36
3.7 Schematic structure of an application with block processing .......... 36
3.8 Lifecycle phases of Implementations ....................................... 37
3.9 Block-size propagation .......................................................... 39
3.10 Block management performed by Units and Implementations ........ 39
3.11 Structure of the critical section ............................................. 40
4.1 Example of decompositions .................................................... 49
4.2 Example of data dependencies ............................................... 52
4.3 Bottleneck component in a parallel stream-processing application .... 53
5.1 DSPE elements of the event-driven infrastructure ....................... 66
5.2 Flow of events between Units ................................................ 67
5.3 Functionality provided by event pools ...................................... 68
5.4 Solution adopted in sendEvent() functions ............................... 70
5.5 Interaction between Unit Behaviours and the infrastructure for events 72
5.6 Transit-dismiss and arm-post supports ..................................... 73
5.7 Lifecycle phases of State Unit Behaviours ................................ 78
5.8 Interaction between Composites and the infrastructure for events .... 79
5.9 Solution supporting different block sizes in Event Units ................ 79
5.10 Possible interactions between coexisting schedulers ................... 81
### List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.1</td>
<td>Production of lightweight tasks performed by Coprocessor Units</td>
<td>86</td>
</tr>
<tr>
<td>6.2</td>
<td>Integration of soft coprocessors with the event-driven Infrastructure</td>
<td>88</td>
</tr>
<tr>
<td>6.3</td>
<td>Lifecycle phases of Coprocessor Unit Behaviours</td>
<td>92</td>
</tr>
<tr>
<td>6.4</td>
<td>Shared and exclusive queues</td>
<td>94</td>
</tr>
<tr>
<td>6.5</td>
<td>Internal states of tasks and Coprocessor Unit Behaviours</td>
<td>95</td>
</tr>
<tr>
<td>6.6</td>
<td>Support for automatically managing events and lightweight tasks</td>
<td>96</td>
</tr>
<tr>
<td>6.7</td>
<td>Dependencies of the automatic features for events and tasks</td>
<td>97</td>
</tr>
<tr>
<td>6.8</td>
<td>Work sharing and work stealing</td>
<td>101</td>
</tr>
<tr>
<td>6.9</td>
<td>Work stealing from neighbour soft coprocessors</td>
<td>102</td>
</tr>
<tr>
<td>6.10</td>
<td>Possible interleaving of lightweight tasks</td>
<td>103</td>
</tr>
<tr>
<td>6.11</td>
<td>Pull pressure strategy</td>
<td>106</td>
</tr>
<tr>
<td>6.12</td>
<td>Push pressure strategy</td>
<td>107</td>
</tr>
<tr>
<td>6.13</td>
<td>Execution pressure infrastructure</td>
<td>108</td>
</tr>
<tr>
<td>7.1</td>
<td>Combined use of CPUs and GPUs</td>
<td>116</td>
</tr>
<tr>
<td>7.2</td>
<td>CUDA grids and CUDA blocks</td>
<td>117</td>
</tr>
<tr>
<td>7.3</td>
<td>Integration of the accelerator and the soft coprocessor infrastructure</td>
<td>119</td>
</tr>
<tr>
<td>7.4</td>
<td>Lifecycle phases of GP Unit Behaviours</td>
<td>121</td>
</tr>
<tr>
<td>7.5</td>
<td>Interleaving performed when using CUDA streams</td>
<td>123</td>
</tr>
<tr>
<td>7.6</td>
<td>Asynchronous execution of kernel tasks with CUDA streams</td>
<td>124</td>
</tr>
<tr>
<td>7.7</td>
<td>Heterogeneous work stealing</td>
<td>125</td>
</tr>
<tr>
<td>8.1</td>
<td>UML diagram of the DSPE Base Model</td>
<td>132</td>
</tr>
<tr>
<td>8.2</td>
<td>Contents of an EMF Resource in DSPE</td>
<td>133</td>
</tr>
<tr>
<td>8.3</td>
<td>Example of the structure of a Definition</td>
<td>134</td>
</tr>
<tr>
<td>8.4</td>
<td>Visual representation of a Unit Definition in DSPE</td>
<td>138</td>
</tr>
<tr>
<td>8.5</td>
<td>Visual representation of a Composite Definition in DSPE</td>
<td>139</td>
</tr>
<tr>
<td>8.6</td>
<td>Visual representation of a Configuration Definition in DSPE</td>
<td>143</td>
</tr>
<tr>
<td>9.1</td>
<td>Example of the internal structure of a generator</td>
<td>154</td>
</tr>
</tbody>
</table>
## List of Tables

3.1 Application lifecycle phases ................................. 33
3.2 Additional lifecycle phases of Implementations ............... 37
3.3 List of the managers provided by DSPE ................. 43
4.1 Example of execution-time variations caused by run-time interferences 57
5.1 DSPE elements of the event-driven infrastructure .......... 65
5.2 Automatic features for managing events .................. 77
5.3 Additional lifecycle phases for events .................. 78
6.1 DSPE elements of the soft coprocessor infrastructure ....... 84
6.2 Additional lifecycle phases of Coprocessor Unit Behaviours 93
6.3 Automatic features for managing tasks .................. 97
6.4 Internal variables of the infrastructure for pull execution pressure 108
7.1 Additional lifecycle phases of GP Unit Behaviours ........ 120
8.1 List of Definitions of the DSPE model .................. 136
<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>API</td>
<td>Application Programming Interface</td>
</tr>
<tr>
<td>CIM</td>
<td>Computer Integrated Manufacturing</td>
</tr>
<tr>
<td>CnC</td>
<td>Concurrent Collections</td>
</tr>
<tr>
<td>CPU</td>
<td>Central Processing Unit</td>
</tr>
<tr>
<td>CSP</td>
<td>Communicating Sequential Processes</td>
</tr>
<tr>
<td>CUDA</td>
<td>Compute Unified Device Architecture</td>
</tr>
<tr>
<td>DSL</td>
<td>Domain-Specific Language</td>
</tr>
<tr>
<td>DSPs</td>
<td>Digital Signal Processors</td>
</tr>
<tr>
<td>DSPE</td>
<td>Digital Stream–Processing Environment</td>
</tr>
<tr>
<td>EMF</td>
<td>Eclipse Modelling Framework</td>
</tr>
<tr>
<td>EBNF</td>
<td>Extended Backus–Naur Form</td>
</tr>
<tr>
<td>FIR</td>
<td>Finite Impulse Response</td>
</tr>
<tr>
<td>FPGA</td>
<td>Field Programmable Gate Array</td>
</tr>
<tr>
<td>GCC</td>
<td>GNU Compiler Collection</td>
</tr>
<tr>
<td>GEF</td>
<td>Graphical Editing Framework</td>
</tr>
<tr>
<td>GMF</td>
<td>Graphical Modelling Framework</td>
</tr>
<tr>
<td>GPGPUs</td>
<td>General-Purpose GPUs</td>
</tr>
<tr>
<td>GPU</td>
<td>Graphical Processing Unit</td>
</tr>
<tr>
<td>GUI</td>
<td>Graphical User Interface</td>
</tr>
<tr>
<td>HDL</td>
<td>Hardware Description Language</td>
</tr>
<tr>
<td>HRTF</td>
<td>Head Related Transfer Function</td>
</tr>
<tr>
<td>ICIMSI</td>
<td>Institute CIM for Sustainable Innovation</td>
</tr>
<tr>
<td>IEC</td>
<td>International Electrotechnical Commission</td>
</tr>
<tr>
<td>IEEE</td>
<td>Institute of Electrical and Electronics Engineers</td>
</tr>
<tr>
<td>IIR</td>
<td>Infinite Impulse Response</td>
</tr>
<tr>
<td>JET</td>
<td>Java Emitter Templates</td>
</tr>
<tr>
<td>KPN</td>
<td>Kahn Process Networks</td>
</tr>
<tr>
<td>MIMD</td>
<td>Multiple Instructions Multiple Data</td>
</tr>
<tr>
<td>Abbreviation</td>
<td>Full Form</td>
</tr>
<tr>
<td>--------------</td>
<td>-----------</td>
</tr>
<tr>
<td>MoC</td>
<td>Model of Computation</td>
</tr>
<tr>
<td>MPI</td>
<td>Message Passing Interface</td>
</tr>
<tr>
<td>OMG</td>
<td>Object Management Group</td>
</tr>
<tr>
<td>OOP</td>
<td>Object Oriented Programming</td>
</tr>
<tr>
<td>OpenCL</td>
<td>Open Computing Language</td>
</tr>
<tr>
<td>OpenMP</td>
<td>Open Multi-Processing</td>
</tr>
<tr>
<td>OS</td>
<td>Operating System</td>
</tr>
<tr>
<td>PGAS</td>
<td>Partitioned Global Address Space</td>
</tr>
<tr>
<td>PThreads</td>
<td>POSIX Threads</td>
</tr>
<tr>
<td>RAM</td>
<td>Random Access Memory</td>
</tr>
<tr>
<td>SDF</td>
<td>Synchronous Data Flow</td>
</tr>
<tr>
<td>SDK</td>
<td>Software Development Kit</td>
</tr>
<tr>
<td>SGMS</td>
<td>Stream Graph Modulo Scheduling</td>
</tr>
<tr>
<td>SIMD</td>
<td>Single Instruction Multiple Data</td>
</tr>
<tr>
<td>SPMD</td>
<td>Single Program Multiple Data</td>
</tr>
<tr>
<td>SSE</td>
<td>Streaming SIMD Extension</td>
</tr>
<tr>
<td>SUPSI</td>
<td>University of Applied Sciences and Arts of Southern Switzerland</td>
</tr>
<tr>
<td>SysML</td>
<td>Systems Modelling Language</td>
</tr>
<tr>
<td>TBB</td>
<td>Threading Building Blocks</td>
</tr>
<tr>
<td>UI</td>
<td>User Interface</td>
</tr>
<tr>
<td>UML</td>
<td>Unified Modelling Language</td>
</tr>
<tr>
<td>WinThreads</td>
<td>Windows Threads</td>
</tr>
<tr>
<td>XMI</td>
<td>XML Metadata Interchange</td>
</tr>
<tr>
<td>XML</td>
<td>Extensible Markup Language</td>
</tr>
</tbody>
</table>
Applications that simultaneously acquire data, modify the obtained information, and instantaneously provide results are said to perform real-time stream processing. Streams are sequences of data items, which may be unordered and unpredictable. For example, stream items can be images, audio samples, temperatures, and pressure values acquired by sensors or received from the network. Many types of streams are possible; however, in most situations, streams are time series of identical type values captured at uniform time intervals. Most streams are unbound, are dynamically produced by unforeseeable sources and, therefore, need to be processed on the fly. Besides, stream processing is useful when offline processing is precluded. For instance, in embedded systems, most computation-intensive applications perform stream processing because of limited availability of storage (e.g., RAM or disk).

Digital signals of the multimedia and telecommunication domains are representative types of streams. In most of the situations, processing of these discrete-time sampled signals requires high throughput and low response latency. Therefore, they are mainly processed with real-time stream-processing approaches. The commonly exploited solution is to let the signal samples flow through software networks of digital filters (see Figure 1.1), which modify the data and extract results on the fly (for more information, see the next section and Section 1.3). Besides the field of digital signal processing, stream-based technologies are frequently used.

Figure 1.1: Example of networked digital filters used in signal processing applications.
in networking systems, and for digital control in domains such as robotics and industrial automation. Furthermore, to spare resources and allow large interactive models, real-time stream processing is also frequently used for simulations—for example, in biology, medicine, mechanics, and material sciences. A chronological overview of stream processing is presented in an article by R. Stephens [127]. Similarly, W. A. Najjara et al. [102] review the utilization of data flow graphs as stream-processing models.

This chapter provides an overview of the common features of stream-processing applications and presents the most popular models of computation and parallel models utilized for stream processing. This chapter discusses real-time stream processing from a conceptual perspective. Technical aspects of the DSPE stream-processing architecture are provided in Chapter 3, whereas details about the parallel stream-processing support are provided in Chapter 4, Chapter 5, Chapter 6, and Chapter 7.

1.1 Main Characteristics of Stream-Processing Applications

Real-time stream-processing applications have common behaviour. They are

- **Computation-intensive**: Streams may be of large size, and stream processing may be time consuming.

- **Time-critical**: Most streams have to be processed continuously, in the order stream items arrive and by respecting strict constraints on processing latency. Results need to be provided immediately, without interruptions while meeting the deadlines.

- **Interactive**: Users often need to modify the behaviour of the application at run time, by providing configuration parameters.

- **Fail-safe**: Applications need to tolerate the presence of corrupted data or errors arising when processing.

- **Able to proficiently use resources**: Poor utilization of resources, such as memory or processors, may compromise the results of stream processing. In particular, to efficiently utilize modern chip-level multiprocessors, applications need to take advantage of possibilities of locality of processing and parallelism available in the algorithms.

These are the behavioral features on which DSPE focuses with most care. An exhaustive review of the requirements of stream-processing applications is provided in an article by M. Stonebraker et al. [129]. For example, the article also analyses the needs of stream-processing applications that retrieve information from databases. Similarly, S. Bhattacharyya et al. [9] provide further details about the requirements of stream processing.
1.2 Common Features of the Application Design and Architecture

Real-time stream-processing applications share common behaviours. Therefore, application designs and software architectures are also similar. This section provides a summary of the most prominent common functionalities of real-time stream-processing applications.

As a general procedure for real-time stream processing, most applications load a portion of the stream into a memory buffer (see Figure 1.2). This sliding window over the stream is mostly needed because of data dependencies but is also often used as a cache for speeding up processing. To fulfil real-time requirements or memory size limits, the stream buffer normally needs to be small.

The high-level design of stream-processing applications is typically composed of modules encapsulating processing algorithms like digital filters. Modules are linked with one another into processing engines, by means of communication channels used for transferring stream items and other kinds of information between modules. This type of approach favours locality of processing, source code isolation, and reuse of algorithms. Several methods, such as object-oriented programming, are available for developing and combining modules into stream-processing applications. Often complex and recursive combination of modules is needed. Therefore, adequate software technologies for application run-time infrastructures and software architectures have to be used.

Processing engines are the heart of stream-processing applications (see Figure 1.3). The software architecture of stream-processing engines and of all surrounding parts requires a throughput-oriented behaviour: to maximize performances and to satisfy real-time needs, it is necessary to minimize interruptions, wait-states, and other overheads, which may slow down the work of the stream-processing engines. Therefore, exchange of information with interactive user interfaces and other external actors is mostly performed through fast critical sections, which provide synchronization functionality without significant interferences. Similarly, concurrent threads of execution, which may be used internally by the processing engines—for instance, for parallel processing—also need special-purpose lightweight infrastructure for data exchange. More information about critical sections and thread synchronization infrastructures that are provided by DSPE is available in Section 3.22 and Section 4.11.

Memory regions and queues used in critical sections and for other synchronization purposes increase the amount of data temporarily stored in the application’s processing buffer required for data dependencies and for data caching. Consequently, parallel real-time stream-
processing applications become complex buffering systems. The size of all buffers and their management strongly impact the application’s latency. If the latency caused by buffering and processing exceeds the limits, the application’s real-time behaviour may be compromised, and time-critical requirements may become impossible to satisfy. Therefore, design and implementation effort is required to keep the application buffer sizes minimal.

Additionally, the following situations require special-purpose support in stream-processing applications:

1. *If streams are incoming from different sources*, like camera, lasers, and infrared sensors, and if they are of different data types. For example, in multimodal signal processing [133, 19, 7], more than one stream originating from the same physical reality needs to be processed simultaneously.

2. *If the incoming or internal data streams have variable rates*. Sometimes, streams have irregular flows. Furthermore, applications may require internal modification of the sample rate as part of their processing. Therefore, for preventing unnecessary inefficiencies, the application needs to dynamically adapt the execution.

3. *If control tasks have to execute side by side with processing tasks*. Frequently, stream-processing applications acquire streams, compute results, and contemporaneously react to external stimuli or emit messages to control external devices or other parts of software. In many situations, the execution priority of control tasks has to be higher than the execution priority of processing tasks. However, their execution should not compromise stream processing.

4. *Situations where adaptive processing is performed*. If recent results are used as coefficients of subsequent processing rounds, like in neural networks, the application run-time behaviour varies significantly, depending on the produced results.

5. *If the application behaviour has to be monitored*—for example, if execution profiles for benchmarking and debugging are needed.
All the required features may substantially raise the complexity of application development. Therefore, adequate methodologies for integrating and managing all application functionalities and all crosscutting features are needed.

1.3 Models of Computation and Parallel Models

Most of the high-level semantics used in stream-processing solutions, for example in the cited stream-processing environments or in programming languages, such as StreamIt, ACOTES, and FastFlow are based on a model of computation (MoC) like the Kahn Process Networks (KPN) or the Synchronous Dataflow (SDF) models. DSPE also exploits the concepts of these MoCs. As will be discussed in this section, the academic literature presents many variations of these models. In most of the MoCs, graphs composed of vertices and arcs (see Figure 1.4) are utilized for describing the applications' high-level structures. Similarly, in the literature, there are descriptions of parallel programming models that are useful for parallel stream processing. These are, for instance, Communicating Sequential Processes (CSP) and Parallel Objects. DSPE also takes advantage of the concepts of these models as part of the DSPE domain-specific language and the DSPE-generated source code.

Some of the most representative MoCs and parallel programming models are the following:

**Kahn Process Networks** [69, 51, 139] is a model frequently used in the domain of digital signal processing. In this model, the vertices of the graphs represent deterministic algorithms used for processing the data streams. Arcs are instead connections between the vertices and represent unbounded FIFO queues. These queues are utilized for exchanging the stream items. The vertices, which mostly represent filters, may be executed serially or in parallel with the goal of processing an infinite data stream either sequentially or concurrently. In this model, if stream items are not available at the inputs of the vertex, the execution of the vertex blocks. Instead, writing to an output queue is always non-blocking. The verification of the availability of data items at the inputs of a vertex is not possible. Visual modelling languages, such as Matworks Simulink and Labview utilize derivations of the KPN model.

Figure 1.4: Example of a graph composed of vertices and arcs.
Chapter 1. Application Family: Real-Time Stream Processing

**Synchronous Dataflow** [86, 85, 45, 110, 10] is a model similar to KPN. Arcs are used for representing FIFO queues between vertices. Vertices have instead a different behaviour. In SDF, firing rules have to be specified, which allows executing the filters only if some condition is met. Firing rules are defined on the basis of the number of stream items available at the inputs of the vertex. In SDF, the number of stream items required for processing and the number of produced outputs are known before execution time. Therefore, SDF is advantageous because many properties of the network, such as the size of the FIFO queues and the execution rates of the vertices, can be determined statically. Furthermore, as discussed in the thesis by M. I. Gordon [54], the scheduling of the execution of the vertices can be decided at compile time. For these reasons, SDF is mostly used in compilers that statically assign the application processing partitions to the cores (see Section 4.12 for more details).

**Petri Nets** [112, 101] is a model used for describing asynchronous and concurrent behaviours in many application domains, such as in distributed systems, automation systems, and networking. Petri Nets are made of two types of vertex transitions, which represent events, and places, which represent preconditions and post conditions of the transitions. Arcs are used for connecting a place with a transition or a transition with a place. It is not possible to connect two transitions or two places. Each place has an associated state, which represents the number of available stream items, called tokens. If a sufficient number of tokens are available, a transition may fire and consume the tokens. Firing is non-deterministic in Petri Nets. Transitions are not forced to fire even if they can. The Petri Nets model is generic and can be used for modelling the execution of many other models, including KPN and SDF.

**Active Objects** [63, 120, 93, 66] is a model utilized in multi-threaded applications to allow concurrent execution of objects. The Active Objects model is also considered a behavioural design pattern [121]. In this model, objects are active because they execute with an independent thread of execution. In order to simplify the access to shared resources, the Active Objects model decouples the method of execution from the method of invocation. Active Objects are also frequently named actors or concurrent objects.

**Parallel Objects** [111, 39, 105] is a model used for object-oriented programming of parallel applications. This model combines the effectiveness of object-oriented constructs with solutions that allow parallel and distributed processing. This model allows both inter-object and intra-object parallelism by providing different semantics for method invocation. As a consequence, it is possible to distribute objects as computing elements on the parallel processing resources. Resource descriptions are used to assign the objects to the available resources. The Parallel Objects model is supported by the POP-C++ programming language.

**Communicating Sequential Processes** [64, 65, 14] is formalism used for describing concurrency in parallel and distributed systems. CSP allows specifying independent processes that interact by means of message passing (see Section 5.1 for more details). CSP allows the parallel execution of any type of processing, including non-deterministic algorithms, by communication via rendez-vous. CSP is supported by programming languages such as Occam and
1.3. Models of Computation and Parallel Models

Go.

**Actors** [63, 25, 2, 3] is a formalism similar to CSP used for describing the concurrency present in programs and systems. As the main unit of computation, the Actors model focuses on independent entities called actors, which communicate by using messages. On the basis of the received messages, actors are free to perform any action including creating other actors, sending messages, and influencing the behaviour they’ll have for subsequent messages. The Actors model was inspired by the programming languages Lisp, Simula, and Smalltalk and has the goal of targeting systems with hundreds or thousands of processing cores. Many programming languages support the actors model, including Erlang and Scala.

**Concurrent Collections (CnC)** [17, 76, 16] is a model that should not be confused with the concurrent collections provided by Java. The CnC parallel programming model is built on past work of TStreams [79, 77, 78] and derives its concepts from streaming and dataflow models. Applications are specified by means of CnC graphs, in which high-level application-specific operations are combined. CnC supports task and data parallelism. The CnC Parallel Programming Model has been implemented both for TBB and for X10.

**Algorithmic Skeletons** [26, 53] is a model utilized for developing pre-configured parallel-processing solutions in form of reusable and configurable patterns. Skeletons hide communication and synchronization operations required for parallel processing and, therefore, facilitate parallel programming. As presented in a publication by M. Cole [26], many frameworks and libraries for Algorithmic Skeletons exist.

Concerning specific models for stream processing, an interesting contribution was provided by A. Navarro et al. [103, 104] with an analytical model for pipeline parallelism based on the networking theory. With the purpose of evaluating the model, a comparison of a static partitioning approach with a dynamic partitioning approach based on work-stealing was performed. A. Navarro et al. utilized PThreads and TBB as infrastructures for parallel processing on an SMP computer.

As will be discussed in more detail in Section 3.1, the applications designed and generated by DSPE are composed of functional elements, which feature processing, communication, and scheduling functionality specific for stream processing. Like KPN, SDF, and CnC, in the stream-flow model of DSPE, the vertices represent the algorithms of filters and other types of processes. However, as in CSP and in the Actors model, the vertices are not restricted to deterministic algorithms or processes.

Unlike in SDF, in DSPE, simple types of connections may also be used. Arcs of the graph may represent shared-memory regions (see Section 3.9) or FIFO queues (see Section 5.3) depending on the specified application design. Similarly to SDF, in DSPE, it is possible to conditionally execute the algorithms associated with the vertices depending on the availability of stream items at the inputs of the vertices. This behaviour is exploited in the DSPE-generated infrastructures to automatically manage events and lightweight tasks (see Section 5.10 and...
Chapter 1. Application Family: Real-Time Stream Processing

Section 6.10 for more details). Like SDF, DSPE supports multiple sample rates by combining Composites and block processing (see Section 3.21 for more details). Furthermore, DSPE also supports dynamic sample rates, which are allowed by event-driven scheduling (see Chapter 5 for more information).

Like Parallel Objects, CSP, and Actors, the DSPE stream-flow model is mostly used for describing the concurrency of programs. Furthermore, similarly to these models, in DSPE, it is possible to use messages for exchanging information between independent processing elements. On the other hand, in DSPE, the vertices of the graphs may behave both as passive and active software components; therefore, it is possible to use them similarly as Active Objects and Actors.

As in the Parallel Objects model, in DSPE, the effectiveness of object-oriented constructs is utilized for parallel programming. However, as described in Section 3.4, in DSPE, objects are implemented in C and generated by the DSPE source-code generators. Like Algorithmic Skeletons, DSPE generates source code on the basis of configurable templates that allow extending the functionalities of DSPE (see Section 2.7 for more details). Furthermore, as with Algorithmic Skeletons, the DSPE-generated source code hides the communication and synchronization operations required for parallel processing.
2 Domain-Specific Development Techniques

Domain-specific development techniques, such as component-based and model-driven generative programming, bring many advantages to the design and implementation of domain-specific applications. These techniques reduce the time needed for producing software prototypes and ease adjusting functionalities during application refinement and customization. Domain-specific development techniques may mask the complexity of low-level implementation details concerning the hardware platform and operating system used. Furthermore, they are very useful when adapting application functionalities to limited computing resources.

This chapter introduces the domain-specific development techniques used in DSPE. Advantages and limits are discussed. Further information about the utilization of these techniques is provided in the chapters that follow. In particular, Chapter 8 describes the idea behind the DSPE model, and Chapter 9 describes the main functionalities of the DSPE development environment.

2.1 Rapid Prototyping and Software Customization

Often, the development of software products demands compromises when fulfilling application requirements. Different forces drive the application's design and implementation in directions such as flexibility, performance, and safety. However, these forces may lack synergy. For example, requirements of configurability frequently fight with needs for efficiency and robustness. In some situations, requirements may even collide, resulting in terms that are impossible to negotiate satisfactorily. These implications are particularly important in software sectors where applications need to operate under strong constraints in terms of high performance and time-critical needs. In these contexts, application development may be hard and may require specialized skills concerning domain-specific algorithms, hardware architectures, and operating systems. By using high-level languages and automatic code generators, it is possible to manage this complexity and to accelerate software development by profiting from rapid prototyping and software customization methodologies. Consequently, it is possible to reduce development costs and application time-to-market.
Chapter 2. Domain-Specific Development Techniques

For these reasons, domain-specific development techniques are frequently used in the software industry in the context of software product lines \([24, 35, 82, 142, 100]\). By taking advantage of product family engineering approaches, in a software products line, a specific product platform based on common underlying application architecture is produced. Commonality as well as variability of the application family are planned and managed. Then, product variants are derived. The distinguishing characteristic over traditional development approaches is a predictive—instead of an opportunistic—reuse of software. Domain-specific development techniques are also used in the methodology of software factories \([1, 60]\). Software factories concentrate on tools, processes, and disciplines for automating the production of software in an industrialized way. For each software factory, a development environment supporting a specific type of application is constructed, with the goal of favouring rapid development of derived application versions. The ultimate goal of software product lines and software factories is mass customization of software \([83, 81]\). As highlighted in a book by C. R. Boër and S. Dulio \([12]\), mass customization should not be confused with the process called assemble-to-order, where products are built by combining product features selected by the consumer. Instead, in mass customization, tailor-made products are produced by exactly fitting the customer's needs.

As discussed in Chapter 1, the development of real-time stream-processing applications can be very challenging. Requirements are present that may be hard to satisfy. Therefore, domain-specific development techniques are well suited for their development. By using high-level abstract models and source code generators, it is possible to take advantage of prototyping and customization techniques to rapidly develop families of stream-processing applications. As will be highlighted in Section 2.5, these approaches are particularly useful when designing and implementing multi-core and many-core applications.

In DSPE, the development process shown in Figure 2.1 is used. During the prototyping phase, available software components are assembled and modified to produce incomplete, working versions of an application. Prototypes allow testing the application's functionalities at early development stages and, if used as demo versions, allow showing the application to stakeholders for gathering feedback (see Figure 2.1). The application design may be consequently modified for fitting newly introduced or reviewed requirements. Then, as soon as a mature version of the application is available, the phase of refinement and customization starts. During this phase, prototypes are optimized and finalized. Additionally, many variants of the application may be produced for fitting particular user and hardware needs. DSPE supports application prototyping and customization at model level. This solution significantly reduces the development effort. Furthermore, DSPE provides special-purpose development infrastructure for facilitating application design and implementation. More details about the prototyping and customization support featured by DSPE are available in Chapter 9, where associated development advantages are also described.
2.2 Model-Driven Generative Programming and Complexity

Model-driven development [8, 125, 46, 35] is a technique that focuses on using high-level models for the design and implementation of applications. In model-driven development, models are used to store abstract information, which is managed and utilized by automatic procedures for creating and evolving applications. The Model Driven Architecture [124, 97, 75, 15] is a style of model-driven development proposed by the Object Management Group (OMG). Today, software infrastructures for model-driven development are provided by the most prominent development platforms. These are, for instance, the modelling tools for Eclipse [44] and the DSL Tools for Microsoft Visual Studio [29].

Model-driven development is often combined with generative programming, a software development approach focusing on automatic processes for source code generation. Formally introduced in a book by K. Czarnecki and U. W. Eisenecker [36], generative programming significantly evolved during the 2000s. An overview of generative software development is
Chapter 2. Domain-Specific Development Techniques

provided in an article by K. Czarnecki [34]. Its origins come from compiler technology, but today it is extensively used in domain-specific tools. Template languages are the most common solutions used for specifying the model-to-text transformations needed for generative programming. In each template, the static parts of the transformation coexist with encapsulated logic, which queries a model and produces the dynamic source code parts.

Model-driven generative programming is of help when dealing with development complexity. By managing the variability of the application family at high abstraction levels in the model, it is possible to select application functionalities and accepted compromises, before source code generation takes place. As a substitute for traditional configuration of applications, which often introduce performance and weaknesses in robustness, high-level models may be used for producing any number of tailor-made applications that directly fit user and hardware needs. Therefore, resulting applications are tuned to the minimal required set of functionalities, with potential consequent benefits in terms of performance and robustness.

2.3 Component-Based and Pattern-Based Code Generators

With component-based development [130, 62, 140], structural and behavioural parts of code are managed as black boxes and combined for composing software applications. Software parts are components if they fully encapsulate functionality, if they agree to standard communication interfaces, and if they are independently deployable. As a clarifying example, objects developed with an object-oriented programming language are software components only if they fulfil the cited requirements.

Typically, software components are combined into software architectures, by means of design patterns [50, 141, 18, 121]. Design patterns are structural and behavioural features of software, which may be replicated for solving recurring problems. Normally, complex software architectures use more than one pattern or variation thereof, in overlap.

In combination with model-driven generative programming, patterns and components manifest further qualities, which, otherwise, do not emerge in handwritten code. As a consequence of their properties of reproducibility and modularity, patterns and components are well suited for programmatic and systematic production of source code. Furthermore, as mentioned before, patterns are often intersected with one another to combine many components. Replicated aggregates of patterns and components may be hard to produce in handwritten programs. Generators are of great help in these situations. By collecting the abstract information of patterns and components in high-level models, leaving the responsibility of producing the source code to generators, patterns and components become easily mouldable.
2.4 Aspect-Oriented Domain Modelling

Aspect-oriented software development [47, 117, 142] is a programming paradigm focusing on separation of concerns during software design and implementation. Aspects are specific crosscutting features, which impact the structure of the source code in a non-localized way.

Normally, software architectures combine many functionalities in complex source code. With general-purpose programming languages, independent handling of orthogonal features can be difficult. Functionalities may be distributed in the code and may share dependencies with one another. Proposed solutions for aspect-oriented programming (e.g., AspectJ [71]) solve the problem by extending existing programming languages with special-purpose syntax. The information on crosscutting concerns is specified independently from the rest of the program. Then, an enhanced compiler does the job of distributing and combining application features.

If model-driven generative programming is used, aspect-oriented techniques, employed at model level, provide an alternative solution. The information on the different aspects may be edited in an independent and localized way at a high level in the model. Then, generators will do the complex task of distributing and combining source code parts of the different functionalities. This technique is part of the method of aspect-oriented domain modelling [58, 59, 20].

2.5 Domain-Specific Techniques and Parallel Programming

Parallel programming is always a complex task, not less so of real-time stream-processing applications, in particular, if high throughput and low latency are contemporaneously required. For these types of applications, domain-specific development techniques reduce the parallel programming difficulties. For example, synchronization between many parts of concurrently executing code may be hard to perform in handwritten programs. Robustness of code generators helps to limit risks of errors.

Furthermore, parallel programming is notably application specific. Available parallelism is hidden in application algorithms and software architecture, and an effective exploitation may differ from application family to application family. Essentially, a generic solution to the problem of parallel programming, such as a universal parallelizing compiler, is very hard, if not impossible, to develop. An approach well suited for a group of applications may be totally inadequate for others. Therefore, domain-specific development techniques are of advantage for parallel programming. Dedicated automatic solutions, featuring well-defined parallel processing solutions, may be very effective for developing individual application families.

DSPE features domain-specific functionalities for designing and implementing parallel real-time stream-processing applications. For instance, in DSPE, component-based development is used for separating the application into connected modules and for explicitly disclosing some forms of parallelism. By using generators, it is possible to systematically exploit this
parallelism. Likewise, the DSPE domain-specific model features abstract information for combining and configuring dedicated parallel design patterns [96, 108, 95, 121]. For each developed applications, these patterns are replicated and adjusted automatically by the DSPE generators.

2.6 Domain-Specific Languages

In DSPE, model-driven, component-based, pattern-based, aspect-oriented, and generative-programming techniques have been combined to form a visual and extensible DSL. The complete development process—including design, prototyping, refinement, optimization and customization—is supported, by means of editing surfaces and assisting functionalities (see Chapter 9).

DSLs [48, 138, 116] are considered fourth generation programming languages, because they have a high abstraction level. DSLs are very near to the application domain and are independent from hardware (see Figure 2.2). Most DSL functionalities have a special purpose, and the producible application variants are limited to a well-defined application family. These kinds of language properties have both advantages and limits.

General-purpose programming languages, such as C and Java, are generic, so they are useful in most application domains. This is a consequence of the abstraction level that these languages have in the transformation from a domain-specific concept to machine code. Software architectures are specific for application families and have a higher abstraction level than

![Diagram](image-url)
2.7. Extension Support in the DSPE Model and Generators

general-purpose programming languages. They are close to a domain-specific concept, which is a solution to a domain-specific problem, and are far from machine code and hardware. DSLs featuring abstract information used for generating software architectures have an even higher abstraction level.

In particular, with general-purpose programming languages, as soon as the amount of complexity crosses a particular threshold, poor solutions in terms of efficiency, robustness, and maintainability may result. Alternatively, DSLs may be used to drive code generators. If the production and optimization of structural and behavioural parts of applications are automated, development responsibilities consistently decrease and the threshold of manageable complexity increases.

There are further advantages. Most DSLs are used to produce tailor-made applications that exactly fit user and hardware needs. For instance, if hardware resources differ from one execution environment to another, a group of applications produced with a DSL may be effective as a replacement of a traditional single configurable application. Additionally, DSLs produce source code systematically and automatically from abstract information, consequently bringing benefits to the application performances and robustness. Application scalability may also have benefits. If application functionalities need to be augmented, for instance, to satisfy an increase of available hardware resources, DSLs may help to expand the application without introducing inconsistencies.

On the other hand, if DSLs are not flexible enough, they can excessively limit the producible application family. Furthermore, the utilization of only abstract information at the DSL level may sometimes drastically reduce fine-tuning and optimization possibilities. In these situations, use of a general-purpose language proves more proficient.

According to the analysis provided by M. Fowler [48], the DSPE DSL is an external DSL with, in principle, a stand-alone nature. However, to enhance its flexibility and to facilitate fine tuning and optimization, the DSPE DSL is extensible. As it will be explained in the next chapter, DSPE takes advantage of a hybrid approach, which allows combining the DSL and general-purpose programming languages. For example, in DSPE, the algorithms of software components can be specified in C, C++, or assembly languages.

2.7 Extension Support in the DSPE Model and Generators

To overcome the DSL limits described in the previous chapter and to allow fulfillment of a wider range of user and hardware needs, DSPE features DSL extension support. With this solution, it is possible to specialize DSPE to vertical application domains as required. In DSPE the following extension possibilities are provided:

Development of new software components: DSPE features this solution, which represents one of the most important forms of DSL extension and specialization. For instance, by
integrating audio, image, and other domain-specific algorithms, DSPE was specialized to vertical application domains such as audio processing and simulation in material science. For the development of software components, DSPE provides model-based code templates. These templates feature the skeleton of typical component behaviours that can be configured at the DSL level and then generated. The variety of available templates provides a flexible solution for developing new algorithms and integrating preexisting source code. For more information see Section 3.16 and Section 8.16.

**An extensible type system:** DSPE features functionality that allows defining new data types by specifying constraints and other type characteristics, such as default values. In many application domains, special-purpose data types may be used for representing and managing domain-specific values. The provided advantages are facilitated development and more robust resulting applications. In DSPE, consistent use of custom-made data types is verified at model level by the validation support. Furthermore, in the generated source code, dedicated infrastructure is produced for automatically testing the constraints and for correcting error-prone behaviours. Section 3.9 and Section 8.8 provide more information about this topic.

**Out-of-the-ordinary application evolutions and customizations:** These types of possibilities are featured by DSPE with model aspects and source code extension points, such as managers. They are isolated and centralized regions in the model and in the code that allow modifying application functionalities for fitting particular user and hardware needs. For example, in DSPE, it is possible to tune the parallel processing behaviour of generated applications by customizing a set of aspect-localized configuration parameters. Then, generators take care of distributing the required changes. Similarly, by using managers, it is possible to substitute threading and synchronization libraries with limited impact on the application source code. Details about DSPE model aspects and source code extension points are provided in Section 8.6 and Section 3.23.

As previously introduced, many of these extensions are possible because DSPE allows a hybrid development approach. In DSPE, it is possible to combine the DSL functionalities with those of the general-purpose programming languages C and C++. Integration with source-code libraries, frameworks, and other preexisting source code is directly allowed at model level. More information is provided in Chapter 8.

The described extension support is provided by DSPE directly as part of its functionalities. However, DSPE is open source and has been developed with the Eclipse modelling tools [44]. If needed, extension of DSPE is also possible, by modifying and extending its source code. For example, one way to extend DSPE is by developing generators that use the DSPE model and that support, for instance, specific hardware platforms or specific software frameworks. For more information on the specific infrastructure provided by DSPE for generators, see Section 9.3. Furthermore, more advanced extensions are possible by also modifying the DSPE model. This type of possibility allows transforming DSPE in a tool that fits even more development requirements and domain specificities. However, as it will be explained in the
next chapter, development of a model and generators is not an easy task.

2.8 Development Challenges of Domain-Specific Languages

The development of a DSL based on model-driven generative programming faces many challenges. These challenges mix the difficulties present when developing members of the DSL's application family, with obstacles introduced by language abstraction, and automatic code generation. Adequate development infrastructure is need.

Furthermore, as previously introduced, DSLs should provide sufficient flexibility for developing a wide range of applications and carefully configuring their functionalities. Syntax and semantics of the DSL should allow fine-tuning and optimization without compromising the DSL abstraction level and domain specificity.

On the other hand, code generators should be able to produce essential, robust, and optimized source code. The generators of a DSL perform a transformation, which may be compared, on a different abstraction level, with the transformation performed by compilers of general-purpose programming languages. If a compiler is not able to produce optimized source code, or is not able to scale well when the program size increases, it is of limited usefulness. The same applies to source code generators. Development of generators faces the challenges of automatic, systematic and programmatic production of source code. The conception of effective source code that is simple to generate and able to scale well as application complexity increases, is not trivial.

In DSPE, particular care was taken to satisfy these needs and provide a solution for designing and implementing a wide application family. The details of the syntax and semantics of the DSPE DSL are provided in Chapter 8, which additionally provides more information about the subject of DSL development. The infrastructure used for developing the DSPE generators is instead described in Section 9.3.
DSPE is a development environment for real-time stream-processing applications. The DSPE DSL allows designing and implementing applications at a high abstraction level (see Chapter 8 for more details). Then, generators take care of the low-level details, by transforming abstract application descriptions into source code. A large set of functionalities provided by DSPE supports the development of parallel processing applications for multi-cores and many-cores. Nonetheless, DSPE may also be used for developing mono-processor versions of stream-processing applications. These types of applications may be useful in many situations—for example, if processing does not require computation intensity but very low latency instead. If the computational capabilities of a single core are sufficient, the overheads of the parallel processing infrastructure may be saved, with consequent benefits on the real-time behaviour of the application. Therefore, DSPE provides a base software architecture that includes all functionalities useful for developing mono-processor application versions. These base functionalities are the foundation of the parallel processing infrastructure, but may be used independently.

Before diving into the conceptual and technical details of the parallel processing infrastructure in the next chapters, this chapter first introduces the most prominent features of the base software architecture of DSPE. At the beginning, this chapter presents the conceptual view. Then, details about the generated source code are provided. This chapter is important because it clarifies and motivates fundamental aspects of the DSPE approach. Without a clear understanding of the base software architecture, it is hard to comprehend the advanced features for parallel processing and the syntax and semantics of the DSPE DSL.

3.1 DSPE Functional Elements

Conceptually speaking, DSPE applications are combinations of several types of functional elements, which feature processing, communication, and scheduling functionality for stream processing. These functional elements are implemented in the source code generated by DSPE and are available in the DSPE DSL for describing application features. Section 8 introduced
Chapter 3. DSPE Base Software Architecture

Figure 3.1: Functional elements used in DSPE applications.

the DSPE stream-flow model. The DSPE functional elements, which may be used either at the specification level or at the implementation level, are all the parts required for developing applications based on that model.

Following main categories of functional elements are provided by DSPE (see also Figure 3.1):

- **Units** are the software components of DSPE. Units connect and communicate with other Units by means of Gates. Units may be Software Units or Composites. The internal functionality of Units is provided by Unit Behaviours (for Software Units) and by Structures (for Composites).

- **Gates** are used to connect Units with other Units and with Runners. Gates are connected with communication channels, which can be of simple or structured data type.

- **Unit Behaviours** specify the internal functionality of Software Units. Unit Behaviours mainly feature algorithms for the different lifecycle phases of the execution of software components.

- **Structures** aggregate and connect Units. In addition, Structures specify the processing order of Units.

- **Runners** are special-purpose software components that play the role of top-level containers for Structures and Units. Runners are the executable components of DSPE.

This chapter describes how elements are generated and utilized in the source code produced by DSPE. Instead, Chapter 8 provides information about the use of these elements in the DSPE model.
3.2 Programming Languages Used for the Generated Code

The DSPE generators transform information written in the DSPE domain-specific language into C/C++ source code. C and C++ are general-purpose structured programming languages with availability of compilers and development environments for most processors and operating systems. C/C++ compilers produce optimized executables with very compact size. Furthermore, C/C++ programs are portable across execution platforms. Today, C and C++ are the languages mainly employed for developing embedded real-time applications. For these reasons, in DSPE, C and C++ were selected as target languages for generating the source code. In the DSPE-generated code, C has been used for most processing and control functionalities. These include the following:

1. The source code of software components, which allows encapsulating algorithms and executing the different lifecycle phases of stream-processing applications.

2. The source code, which schedules the execution of components and allows their communication.

3. Critical sections for exchanging information with the GUI and other parts of the application without compromising the processing performances.

4. The infrastructure that supports managing memory and threads in a way specific for stream processing.

In DSPE, algorithms encapsulated in software components are also mostly written in C. However, integration of algorithms with C++ and assembly languages is also possible.

C++ is instead used for the GUIs and for all parts of generated code that integrate with third-party C++ frameworks. For example, the source code of GUIs extends the wxWidgets open-source framework [145]. For audio processing, integration with the VST technology by Steinberg provides support for VST plugins (see Section 9.10 for more details).

3.3 Design Patterns Used in the Base Software Architecture

As introduced in Section 2.3, software design patterns are extensively used in the DSPE-generated source code. The following patterns, or variations thereof, are predominantly used in the base architecture:

**Enhanced versions of the pipes and filters pattern** [18, 108]: a pipeline is obtained by chaining black-box processing components (filters) with connections between their inputs and outputs. Buffers are used for exchanging the data between consecutive components. Similarly as in KPN and SDF (see Section 1.3), in the DSPE stream-flow model, the pipeline pattern has been enhanced to allow constructing networks of components instead of only simple chains. Further details about how this pattern has been used in DSPE are provided in Section 3.11.
Chapter 3. DSPE Base Software Architecture

Composite pattern [50]: with this pattern any number of elements of a given type are encapsulated in a container. By means of inheritance, the container shares the same type of the elements and, therefore, can be used in another container together with other elements. The composite pattern favours modularity and reusability. In DSPE, the composite pattern is used for encapsulating Structures in Units, which can then be used inside other Structures. More information about the composite pattern in DSPE is provided in Section 3.13.

Template method pattern [50]: with this pattern it is possible to specify a default behaviour that may be extended and specialized in specific software parts. Normally, template methods profit from inheritance in object-oriented languages. Instead, in DSPE, source code generation is used for generating the skeleton of the software components. See Section 3.16 for more information.

Decorator pattern [50]: this pattern allows dynamic adding of more functionality to a software part. In object-oriented languages, it is mostly implemented by attaching dynamically created objects to other objects. In DSPE, this pattern is, instead, mostly implemented with function pointers. See the next section for more details.

Generation gap pattern [141]: is a pattern that allows customizing generated source code by separating the generated parts from the performed customizations. At any moment, source code may be regenerated without compromising the customizations. In DSPE the generation gap is implemented in C, by utilizing the object-oriented development techniques explained in the next section.

3.4 Object-Oriented Development in C

The C source code generated by DSPE extensively uses structures and pointers, to allow object-oriented programming in C. At generation time, object-oriented programming is useful, because it facilitates the production of the source code from the information contained in the DSPE domain-specific model, in particular for design patterns.

C is not a language with explicit support for object-oriented programming. Its syntax misses reserved words for encapsulation, inheritance, and polymorphism, which are instead present in languages such as C++ and Java. Nonetheless, by combining several programming techniques based on C structs and pointers, it is possible to use object-oriented programming in C. Due to the complexity of this approach, however, its use for programming large applications by hand may be difficult. In this context, source code generators are of help. By systematically generating the combination of C structs and pointers, it is possible to reduce lack of robustness when extensively using these techniques.

The principal techniques for object-oriented programming in C are described in detail in a book by M. Samek [120] and an article by R. Kreymborg [80]. The relevant aspects are below:

Encapsulation (as inclusion, not information hiding): C provides structs as a solution for
aggregating the state of applications. By taking advantage of C structs, it is possible to enclose parts of the application into objects. Similarly, as in C++ or Java, it is possible to instantiate and use these objects in the rest of the program, by allocating the structs as dynamic memory and by using pointers to reference the C structs. Thereby, in an object-oriented terminology, C structs represent classes, and dynamically allocated C structs represent objects. By following this approach, pointers are always used as parameters for exchanging objects in function calls and to store objects in local variables. By means of function pointers, it is also possible to include functions in the structs and, therefore, to add methods to the classes. To behave like standard methods, these functions should have an explicit parameter of the type pointer to the containing object, for emulating the ‘this’ reference of object-oriented languages.

**Inheritance:** in C, it is possible to implement single inheritance between two classes, by taking advantage of the nesting property of C structs. By including a struct of the type of the parent class as the first field of the struct of the child class, the child class automatically contains all the fields of the parent class, consequently extending it.

**Polymorphism:** by taking advantage of some pointer properties, it is possible to use polymorphism in C. The object’s pointer always references the first memory location of the memory area allocated for the object’s struct. If structs of parent classes are always declared as the first field of child classes, it is possible to cast the pointer of a child object to a pointer of the parent type and to use the casted object as if it were of parent type. Consequently, polymorphic use of the object is possible. Simple forms of override and specialization of methods are possible too, by substituting the installed function pointer in the struct. More advanced forms of method override are described by M. Samek [120] and by R. Kreymborg [80]. Examples of the use of these techniques are provided in Section 3.8 and Section 3.12. DSPE uses these techniques with some variations compared to the explanations provided by M. Samek [120] and by R. Kreymborg [80]. The main differences are

1. Function calls are performed both by means of function pointers and by direct function calls. Sometimes, direct calls are more flexible to use. For example, in the DSPE-generated code, only simple forms of method override are used. Function pointers of the overridden methods are not stored in the object’s struct, as explained, for instance, in the book by Miro Samek [120]. Therefore, in DSPE, the overriding functions perform direct calls to the overridden functions as a way to emulate calls to “super”. This alternative approach reveals efficiency when used in combination with source code generation.

2. Dynamic dispatching, which in regular objects is performed by means of overridden methods, is utilized in the DSPE-generated code in a more general way. External source code is allowed to install and override methods of objects too. This solution allows extending and specializing the behaviour of objects from other parts of the program, instead of only from child objects. This solution is used for implementing the decorator
pattern described in the previous section. An example of its utilization is provided in Section 5.5.

3. Virtual support for multiple inheritances is provided with an artifice. Different from the simple inheritance solution described above—which requires the parent struct to be declared as the first field of the child struct—for multiple inheritances some tricks are required. The structs of additional parents are included as additional fields of the child struct too. Therefore, the child struct also extends these additional parents. To support polymorphic use, a pointer in the parent struct is employed for keeping track of the main child struct. Special-purpose define directives for performing cast operations (see Listing 3.1) are used as a replacement of the standard cast operator of C. Furthermore, a variant of this inheritance artifice is possible by including a pointer to the parent struct instead of its struct directly. By assigning the pointer only if needed, this variant allows a form of conditional extension.

Listing 3.1: Extension artifice used to allow multiple inheritance in DSPE

```c
1 struct DSPEExtension {
2     DSPEElement *child;
3 }; 
4
5 struct DSPEAdditionalParent {
6     DSPEExtension extension;
7     ...
8 }; 
9
10 struct DSPEChild {
11     DSPEFirstParent parent;
12     DSPEAdditionalParent additionalParent;
13     ...
14 }; 
15
16 #define PARENT_CAST(element) &element->additionalParent 
17
18 #define CHILD_CAST(extension) ((DSPEExtension*) extension)->child
```

### 3.5 File Organization of the Generated Code

The DSPE generators partition the generated source code in more than a source file. In principle, there is a file or group of files for each DSPE functional element. This approach simplifies model-driven code generation and favours reuse of generated source code. Generated files may be reorganized in projects as needed. The amount of generated files for each functional element is influenced by the values of model Aspects (see Section 8.6). For example, different files are generated for different versions of a Unit, because of the Block Processing Aspect (see
3.6 Class Hierarchy of the DSPE Objects

The source code produced by the DSPE generators is a software architecture that combines patterns made of objects. These objects are implemented with the object-oriented techniques described in Section 3.4. As roots of the C\texttt{struct} hierarchy of all the generated DSPE objects, the generators produce parent abstract types.

Figure 3.2 provides, in UML notation, all parent abstract types of the DSPE base software architecture. These abstract types mostly reflect the structure and behaviour of the conceptual DSPE elements described in Section 3.1.

In the generated source code, abstract types are extended and specialized by concrete C\texttt{struct}s. Then, these \texttt{struct}s are instantiated and used for implementing the functionality of

Section 3.19 and Section 8.11). Then, depending on the application design, Structures use only the source file of the Unit they require. This approach allows Structures with different aspect values to coexist.

Figure 3.2: DSPE elements of the base software architecture.

![Class Hierarchy Diagram]

- **DSPEElement**
  - Element* container
  - Application* application
  - Owner* owner
  - char* (getID) (Element)

- **DSPEComponent**
  - void (*preprocess) (Component)
  - void (*process) (Component)
  - void (*postprocess) (Component)
  - void (*setbypass) (Component, int bypass)

- **DSPEStructure**
  - DSPECConfiguration
  - DSPEImplementation

- **DSPEUnit**
  - DSPEUnitBehaviour
  - DSPEBlockOptimization

- **DSPERunner**
Chapter 3. DSPE Base Software Architecture

stream-processing applications. Polymorphic use of the abstract types is mostly done, with the goal of uniformly managing heterogeneous objects.

All DSPE elements share the common parent DSPEElement, which mainly features functionality for accessing its containers like the direct container and the Runner (Listing 3.2 provides the struct of DSPEElement). Furthermore, each DSPEElement contains a sequence of characters used as a unique identifier of the element. One of the abstract types that is the direct child of DSPEElement is DSPEComponent. It is the common parent of all software components in DSPE. DSPEComponent features functionality for the main lifecycle phases of components (Listing 3.2 also provides the struct of DSPEComponent). Child elements of DSPEComponent are all Units, Composites, and Runners. All Implementations and Block Optimizations are instead child elements of DSPEUnitBehaviour. DSPEStructure is the common ancestor of Configurations, Schedulers, and Coprocessor Schedulers (see Section 5.8 and Section 6.7 for more details about Schedulers and Coprocessor Schedulers). More details about the specific abstract types of the base software architecture are provided in the sections that follow.

Abstract types are also available for the event-driven infrastructure and the parallel processing support. More information is provided in Section 5.2 and Section 6.1.

Listing 3.2: C structs of DSPEElement and DSPEComponent.

```
struct DSPEElement {
    DSPEElement *container;
    DSPERunner *runner;
    DSPEOwner *owner;
    char* (*getID) (const DSPEElement *element);
};

struct DSPEComponent {
    DSPEElement element;
    void (*preprocess) (DSPEComponent *component);
    void (*process) (DSPEComponent *component);
    void (*postprocess) (DSPEComponent *component);
    void (*setbypass) (DSPEComponent *component, int bypass);
};
```
3.7 DSPE Components

There are three main types of software components in DSPE: Software Units, Composites, and Runners.

According to Section 2.3, Software Units and Composites have all the characteristics of software components. They are reusable and independently deployable and, similarly to black boxes, feature encapsulated behaviour. They are used in Structures and exchange information with one another by means of Gates, which are their standardized connection and communication interface.

Runners are instead elements of execution in DSPE. They contain Structures and, consequently, Units and Composites. Runners are compiled and linked to form executables. Therefore, Runners are special kinds of software components, which are not reusable as part of other DSPE components.

3.8 Software Units

In DSPE, Software Units are components that directly feature processing or control functionality by means of algorithms. The algorithms used by Software Units are provided by Unit Behaviours and may be domain specific or general purpose. For example, Software Units may encapsulate either multimedia processing algorithms or simple routines to open and read files.

Listing 3.3: Example of the C struct of a Software Unit.

```
struct Example_SoftwareUnit {
    DSPEUnit parent;
    /* Parameter Gates */
    GateTypeOne *paramIn_FirstInput;
    GateTypeTwo *paramIn_SecondInput;
    GateTypeThree *paramOut_FirstOutput;
    /* Data Gates */
    GateTypeTwo *dataIn_ThirdInput;
    GateTypeOne *dataOut_SecondOutput;
    /* Unit Behaviour */
    Example_BlockOptimization untiBehaviour;
    /* Block size */
    int blockSize;
};
```
Chapter 3. DSPE Base Software Architecture

Typically, Software Units are filters, which perform some type of transformation on the data streams. In other situations, Software Units may be state machines, which control an automation process. Software Units may also contain independent threads of execution or react to operating system interruptions. As a consequence, Software Units may also be active application elements.

All generated Software Units extend the abstract type DSPEUnit, which is associated with common behaviour for unit construction, initialization, execution, and destruction. A simplified version of a sample C struct of a Software Unit is provided in Listing 3.3.

As it may be remarked in Listing 3.3, the struct of a Software Unit directly contains the struct of its Unit Behaviour. Furthermore, it contains information about the block size used for block processing (see Section 3.19 for more details). Gates are, instead, included as pointers, as will be explained in the next section.

3.9 Gates

In DSPE, software components feature Gates, which represent the inputs and outputs used for exchanging information with the outside of the component. Furthermore, in DSPE, Gates behave as standardized connection and communication interfaces, which allow linking components by means of compatible communication channels.

In the source code, Gates are references to memory regions that are used as buffers for exchanging the information between components. Memory regions associated with Gates may be shared and directly accessed by software components or managed by events (see Chapter 5 for more details). In the case of shared-memory regions, as shown in Listing 3.3, pointers are used for referencing and accessing the memory regions. This type of approach is effective, as long as concurrent access to the memory region from more than one thread of execution is not required.

Furthermore, in DSPE, it is possible to utilize Group Gates, which are particularly useful if a Unit requires a variable number of Gates of the same type. For instance, in the audio domain, Group Gates may be used for Units that provide either mono, stereo, or surround audio channels depending on the application in which they are used. The source code generated for Group Gates is implemented by using arrays of Gates.

The memory region referenced by Gates may be of simple or structured data type. Therefore, C standard types, arrays, and C structs may be used as types for Gates. Alternatively, there are Pointer Gates and String Gates that contains pointers. More information about Pointer Gates and String Gates is provided in Section 3.20.
3.10 Parameter vs. Data Gates

In DSPE, Software Units and Composites have 2 types of inputs and outputs: Parameter Gates and Data Gates. Different execution policies and priorities are used for managing these Gates:

For Parameter Gates, instantaneous values are mostly considered. Values may be lost if processing is not able to keep track of all changes. Parameter Gates should be used for exchanging control information, such as values coming from the GUI or from other external devices that are used for configuring algorithms at run time.

For Data Gates, all values are utilized for processing. Data Gates have to be used, if loss of values may result in data corruption. Data Gates imply more overheads than Parameter Gates. Data Gates should be employed for all streams of data.

A distinction between Parameter Gates and Data Gates is made in DSPE, to facilitate real-time processing. The infrastructure produced for Parameter Gates differs from the infrastructure generated for Data Gates. Irregular flows of values may coexist with continuously flowing data streams. As an example, an interactive GUI may produce considerable amounts of values at irregular time intervals. Normally, only the instantaneous value is significant at each instant and value modifications should immediately impact processing. Therefore, for GUI values, Parameter Gates should be used. The infrastructure generated for communicating parameters is designed to avoid slowing down processing or introducing other types of performance degradations. This is particularly important in stream-processing applications, if time-critical requirements have to be respected.

This distinction between Parameter and Data Gates mostly has an impact on the source code produced for block processing and for the critical sections. More details are provided in Section 3.19 and Section 3.22.

3.11 Structures

Structures are containers and managers of Software Units and Composites. All connections between software components are done in the generated code of Structures, which are also responsible for scheduling the execution of components. Within DSPE Structures, as introduced in Section 3.3, networks of interconnected components with feedback loops are allowed. There are three types of Structures in DSPE: Configurations, Schedulers, and Coprocessor Schedulers. They mostly differ in terms of the scheduling approach used and the featured parallel processing support. Configurations are discussed in the next section, Schedulers in Section 5.8, and Coprocessor Schedulers in Section 6.7.
3.12 Configurations

Configurations are the simplest form of Structure. In Configurations, scheduling of components is performed by iterating a fixed execution sequence (see Listing 3.4 for an example). The rest of the infrastructure provided by Configurations is responsible of constructing and connecting components, in agreement with the connection scheme specified at model level.

Listing 3.4: Example of the C struct and process function of a Configuration.

```c
struct Example_Configuration {
    DSPEConfiguration parent;

    /* Units */
    FirstType_SoftwareUnit firstUnit;
    SecondType_SoftwareUnit secondUnit;
    ATypeOf_Composite aComposite;
};

/* Process function */
void process(Example_Configuration *conf) {
    DSPEComponent *component;

    /* Execution of contained components */
    component = (DSPEComponent*) &conf->firstUnit;
    component->process(component);
    component = (DSPEComponent*) &conf->secondUnit;
    component->process(component);
    component = (DSPEComponent*) &conf->aComposite;
    component->process(component);
}
```

3.13 Composites

Composites allow encapsulating Structures within software components. Then, Composites may be used in other Structures in the same ways Software Units are used. Like Software Units, Composites have Parameter and Data Gates in the form of pointers to memory regions. However, in Composites, Gates are mostly used at initialization time with the purpose of connecting the Gates of Software Units. Then, if events are not used, memory regions are directly shared by the Gates of Software Units for exchanging the information, and, consequently, the Gates in Composites are bypassed (see Figure 3.3). For information about the behaviour of events in Composites, see Section 5.12.

Composites favour modularity and reuse of source code. For Composites, the DSPE generators produce a C struct similar to the C struct of a Software Unit, but which contain a C struct of a Structure, instead of a C struct of a Unit Behaviour. Additional connections between the
Gates of the Structure's Units and the Composite's Gates are performed at initialization time. The source code generated for Composites may be rather complex, in particular, concerning the infrastructure for block processing. See Section 3.19 for more details.

3.14 Runners

To allow executing the source code produced by DSPE and to allow interacting with the generated application functionalities, DSPE features Runners. Runners are top-level containers that drive the generated stream-processing engines made of Structures, Composites, and Software Units. DSPE is able to generate three types of Runners:

Command-Line Runners provide simple and limited support for executing the generated applications. It is possible to interact with the applications only with launch-time arguments. Run-time interaction is not allowed. When execution starts, stream processing also directly starts, and it is possible to stop the application only by killing it.

Shell Runners provide textual user interfaces for configuring the application's execution at run time. Therefore, it is possible to interact with the application's behaviour, by means of textual commands. An independent thread of execution is used for stream processing. A critical section, which grants thread safety, is utilized for synchronizing the main application thread and the stream-processing thread (see Section 3.22 for more details).

GUI Runners feature a graphical user interface for managing the application at run time. All functionalities for interacting with the application's behaviour are provided by visual elements, such as buttons, sliders, and check boxes. Similarly to Shell Runners, an independent thread of execution performs stream processing. Synchronization between the main thread and the stream-processing thread is also done with a critical section.

Additionally, with DSPE, it is possible to generate Custom Runners and Empty Runners. In these cases, only partial implementations of the applications are produced. These Runners
allow the development of custom-made user interfaces or the integration of the generated source code with other preexisting applications.

Like Composites, Runners contain Structures. However, Runners may have only Parameter Gates, which are used for connecting the contained Structure with the generated user interface. The main C struct generated for a Runner looks like the C struct generated for a Software Unit. However, depending on the type of Runner and specified application functionalities, for Runners, generation of many other C structs and C++ classes may be performed. Indeed, for Runners, the DSPE generators may produce large amount of elaborate source code, in particular, concerning the critical section.

Figure 3.4 contains the state chart of a Shell or GUI Runner. Depending on the interaction of the user with the application, various types of state transitions are possible. Normally, the
application is in the stopped, running, or paused states. In addition, it may be put in the suspended state by the algorithms, by means of suspend or freeze function call (freezing is more immediate than suspending, but leaves processing incomplete). Furthermore, a special-purpose state for skipping parts of processing is available, and a stopping state is used during the transition between executing and stopped.

3.15 Application Execution: Lifecycle Phases

Depending on the type and behaviour of the Runner and as consequence of the performed state transitions, the stream-processing applications generated with DSPE execute different lifecycle phases. The principal phases are described in Table 3.1 and shown in Figure 3.5.

All Structures, Composites and Software Units feature these five main lifecycle phases. Each phase is implemented in the source code, by means of an independent function generated specifically for each of these DSPE elements. If software components are contained in a Configuration, the execution sequence of these functions is scheduled using the fixed execution sequences specified by the developer. Instead, for all types of schedulers, event-driven scheduling is used for all process functions (see Section 5.8 and Section 6.7 for more details). For the block processing support, the event-driven infrastructure, the soft coprocessor infrastructure, and the accelerator infrastructure, the DSPE-generated code provides additional special-purpose lifecycle phases (see Section 3.19, Section 5.11, Section 6.8 and Section 7.4 for more details).

Table 3.1: Lifecycle phases of DSPE-generated applications.

<table>
<thead>
<tr>
<th>Phase</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Start-up</strong></td>
<td>Executed automatically when the application starts. Used for performing application initialization. After start-up, the application either sits idle waiting for the user to request a run transition from stopped to running, or automatically performs the transition and directly executes the pre-process phase.</td>
</tr>
<tr>
<td><strong>Pre-process</strong></td>
<td>Executed during a transition from stopped to running. Used for preparing the application before processing. After pre-processing, the process phase automatically starts.</td>
</tr>
<tr>
<td><strong>Process</strong></td>
<td>Executed repeatedly when processing. The process phase stops when a transition to the stopped state is requested or if the application is closed. Alternatively, the process phase may be paused or suspended.</td>
</tr>
<tr>
<td><strong>Post-process</strong></td>
<td>Executed when the application is stopped or closed while processing. Used for completion tasks after processing. After post-processing, the application sits idle waiting for a new run or automatically starts the shutdown phase if the application has been closed.</td>
</tr>
<tr>
<td><strong>Shutdown</strong></td>
<td>Executed when the application is closed. Used for application finalization.</td>
</tr>
</tbody>
</table>
3.16 Unit Behaviours

Development of new algorithms and integration of preexisting source code play an important role when implementing Software Units. Special-purpose infrastructure may help, in particular, if functionality in third party libraries and frameworks has to be used. To ease the development of components, DSPE features Unit Behaviours—source code templates that provide ordinary skeletons of the internal parts of components. Unit Behaviours allow encapsulating algorithms in Software Units and provide shortcuts for interacting with the Unit’s Gates and with other functionalities of the Unit. The two types of Unit Behaviours are below:

Implementations are the default type of Unit Behaviours. They mostly feature functions for the main lifecycle phases, shortcuts for accessing to the Unit’s Gates, and infrastructure for managing the internal state of Units. The process function of an Implementation executes for every item of the data streams that traverse the Unit.

Block Optimizations are instead used in combination with the Block Processing Aspect (see Section 3.19 and Section 8.11 for more details). The process function of a Block Optimization executes for every block of item of the data streams. The block is visible in the process phase. Therefore, it can be accessed in any required way.

In addition, in DSPE there are five categories of Unit Behaviours: Simple Unit Behaviours, State Unit Behaviours, Coprocessor Unit Behaviours, GP Unit Behaviours, and Wrap Unit Behaviours. Simple Unit Behaviours provide minimal functionality and are general-purpose. The other categories are specific for the event-driven infrastructure, the soft coprocessor infrastructure, the accelerator infrastructure, and the wrap support (see Section 5.9, Section 6.8, Section 7.4, and Section 3.18 for more details).
3.17. Runner Behaviours

Shortcuts, implemented in C with define directives, are available in Unit Behaviours for accessing the Unit's Gates and other Unit functionalities. Shortcuts provide isolation of algorithms from the rest of the software architecture. In many situations, they allow modifying and regenerating the application without changing the algorithm's implementation. An example of the Unit Behaviour's shortcuts is provided in Listing 3.5.

Listing 3.5: Example of shortcuts available in Unit Behaviours.

```c
1 /* Shortcut for Output Parameter Gate */
2 #define pOut_firstOutput (*context->paramOut_firstOutput)
3
4 /* Shortcuts for Input Data Gates */
5 #define dIn_firstInput (*context->dataIn_firstInput)
6 #define dIn_secondInput (*context->dataIn_secondInput)
```

3.17 Runner Behaviours

Runner Behaviours are similar to Unit Behaviours but are encapsulated and executed by Runners. Differently from Unit Behaviours, they miss a process phase. Therefore, only functions for start-up, pre-process, post-process, and shutdown lifecycle phases are available. Mostly, Runner Behaviours provide a means for global initialization operations and for storing and managing centralized application state. The information stored in Runner Behaviours is accessible from all Unit Behaviours. Therefore, the Runner Behaviour state may be used for sharing global information between Units.

3.18 Wrap Unit Behaviours

Wrap Implementations and Wrap Block Optimizations are a form of Unit Behaviours, which allow programmatic management of Composites—in particular, flexibility in terms of Composite execution. Each Wrap Unit Behaviour features special-purpose infrastructure for allocating any number of instances of an associated wrapped Composite. Therefore, dynamic creation, execution, and destruction of Composites are possible. For instance, a wrapped Composite may be instantiated, used, and disposed for each block of stream-items processed by the Wrap Block Optimization.

3.19 Block Processing

Memory areas, used as buffers for exchanging information between connected components, provide important optimization possibilities. Each time a process function of a Software Unit is called, instead of processing individual stream items as shown in Figure 3.6, it is possible to process a block of items as in Figure 3.7. As a consequence, the execution frequency of components and the overheads required for their scheduling reduce. Furthermore, this
optimization favours locality of processing. In particular, on multi-core processors, with block processing it is possible to take advantage from the available processor caches (see Section 4.6 for more details).

The generated infrastructure for block processing differs between Parameter Gates and Data Gates. Even if block processing is enabled, parameters are always processed as individual items. Therefore, the processing rate of parameters is significantly lower (a fraction of the block size) than the processing rate of the data. In other words, parameters are processed with lower priority. Only the instantaneous value available for the first item of the stream block is considered and used for processing the complete block. This approach is of benefit in terms of performance and real-time behaviour.

In the DSPE-generated source code, block processing has been implemented with arrays. Static and variable block processing are allowed. Whereas in the static version, the block of data has to be completely processed, in the dynamic version, as a way to emulate blocks with variable sizes, it is also possible to process partially filled blocks.

DSPE allows using Implementations and Block Optimizations independently from block processing. If block processing is active, the process functions of Implementations are executed in loops. This solution allows processing all the stream items present in the block. Alternatively, if block processing is not active, the process functions of Block Optimizations are executed with blocks containing only one stream item. These two forms of execution are not optimal, but allow executing the application, even if the exact type of Unit Behaviour is not provided.
3.20 Managed Pointer Gates and In-Place Processing

With Block Optimizations, it is possible to profit from the visibility on the block to optimize the algorithms—for instance, by using smart loops on the complete block. This is not possible in Implementations, where blocks are not visible. As a consequence, if the Block Processing Aspect is active, to avoid repeatedly processing the same parameter values, additional phases are provided in Implementations, as described in Table 3.2 and shown in Figure 3.8.

Table 3.2: Additional lifecycle phases of Implementations.

<table>
<thead>
<tr>
<th>Prepare parameters</th>
<th>Executed once before the process loop. Useful for preparing parameters before block processing.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Finalize parameters</td>
<td>Executed once after the process loop. Useful for finalizing parameters after block processing.</td>
</tr>
</tbody>
</table>

3.20 Managed Pointer Gates and In-Place Processing

Pointer Gates and String Gates are efficient for communicating information between software components because they allow exchanging references of memory regions, instead of directly exchanging the contained information. However, these memory regions may have to be allocated, managed, and disposed. In particular, the critical section and other parts of the generated source code need sometimes to do copies of the values they are transferring. These copies may be required for granting thread safety. In these situations for Pointer Gates and String Gates, a complete copy of the data is required, instead of only a reference copy.
Therefore, DSPE allows specifying if Pointer Gates and String Gates are managed or unmanaged. With unmanaged Gates, the developer is in charge of programmatically allocating and disposing the memory region referenced by the Gates. Instead, with managed Gates, the source code generated by DSPE takes care of memory management by automatically allocating and disposing the required memory regions. As a consequence, with managed Gates, special-purpose functions, such as allocate, copy, and dispose, are required by the infrastructure for each type of Gate. These functions, which are implicit for String Gates, need, instead, to be explicitly specified for Pointer Gates.

Pointer Gates are particularly useful when processing the data stream in-place. With in-place processing, stream items, such as images, are always stored and modified in the same memory region. Only pointers to memory are transferred between Units. In-place processing is an alternative to copy-on-write processing, which is the default approach used in DSPE (see also Section 4.3). Normally, each time an item is processed by a Unit, the item is copied to a new location from an input to an output gate. In some situations, in-place processing may be easier to use, may allow saving resources and may produce better performance. However, whether these advantages are available strongly depends on the type of application. Furthermore, in-place processing needs particular care because risk is high that Units overwrite other Units’ changes. This happens if processing is split in more than one data path or if processing is performed concurrently by more than one thread of execution.

3.21 Composites and Block Processing

The possibility of using different block sizes as part of the software architecture of a stream-processing application is an important functionality featured by DSPE. Use of different block sizes is allowed by Composites and by variable block processing. Some advantages are introduced, such as the possibility of profiting from stream-rate changes or the possibility of executing components with different frequencies. Special-purpose infrastructure, featured, for instance, by the forceCriticalSection() function—its functionality is explained in the next chapter—is of help in terms of real-time behaviour even if some parts of the application execute with very low frequency.

However, different block sizes increase the complexity of the infrastructure generated for block processing. The memory regions used for exchanging information between software components inside and outside of Composites may differ; therefore, some form of reconciliation mechanism is required. In the DSPE-generated code, using the technique of block-size propagation during initialization and pointer artifices during processing solves the problem.

At initialization time, each component is informed of the block size of its connections by means of block-size propagation (see Figure 3.9). Consequently, components know the maximum size of each block associated with their Gates. The block size information is propagated for all links present in the application. Therefore, each component is aware of the size it needs to use when allocating memory regions. As illustrated in Figure 3.9, for correct block-
3.21. Composites and Block Processing

Figure 3.9: Propagation of the block sizes in Composites. 1. the block size is propagated downward for all output Data Gates; 2. the maximum block size is propagated upward to inform all connected output Data Gates; 3. propagation continues horizontally inside the Structure; 4 the maximum block size is propagated downward for all input Data Gates.

size propagation, sub-phases of the start-up lifecycle phase are used. The block size is first propagated downward for all output Data Gates. Afterward, as soon as the maximum block size has been stored for the related output Data Gates, this maximum block size is propagated upward to inform all connected Data Gates. Then, propagation continues horizontally inside the Structure, and finally it is propagated downward for all input Data Gates.

Additionally, when different block sizes are used, pointer artifices are employed for managing the memory regions of Data Gates. After gate-size propagation, the memory region associated with a Data Gate is allocated of the largest required block size. In the Unit Behaviour, this memory area is accessed by means of shortcuts, which are automatically updated to the correct position before executing each process lifecycle phase. As illustrated in Figure 3.10, for instance, if the external block size is 16 and the internal one is 4, a block of 16 is allocated.

Figure 3.10: Block management performed by Units and Implementations.
For each execution of the process phase of the Unit, the shortcut is updated to reference one of the sub-blocks of size 4, whereas for each execution of the Implementation, the shortcut is updated to reference the next stream item. When the end of the memory region is reached, the shortcut is rewound to reference the first position of the block. For this purpose, Software Units store a counter for each Data Gate, which allows correctly moving the shortcut over the data block.

### 3.22 The Critical Section

Shell Runners and GUI Runners provide the infrastructure for monitoring and modifying the application behaviour at run time. This infrastructure uses independent threads of execution for performing stream-processing operations and updates of the user interface at the same time. To allow thread-safe updates of the Runner’s parameters, a critical section is generated by DSPE, which allows user interaction at unnoticeable loss of performance. A scalable solution is provided to accommodate a variable number of Runner parameters.

Configurations and Schedulers update the critical section by using a special-purpose function called `forceCriticalSection()`. This function is called regularly by the generated source code to favour the real-time behaviour of the application. As a result, it is possible to aggregate Configurations and Schedulers in the same application and to use different block sizes, without any compromise in terms of updated frequency of the Runner parameters. Furthermore, it is possible to call `forceCriticalSection()` explicitly from the algorithms inside Unit Behaviours.

To allow low cost updates, a solution with swap areas is used to manage the critical section (see Figure 3.11). As introduced in Section 1.2, low cost updates are particularly important to avoid slowing stream processing. The processing engine and the UI are always free to write information in each respective output swap area. As long as the critical section is not updated, in agreement with the real-time policy introduced in Section 3.10, new parameter values overwrite previously existing ones. When the source responsible for updating the critical section is executed, swap areas are exchanged. Consequently, output swap areas become input swap areas, and the parameter values can be read from the other side of the critical section.

![Figure 3.11: Structure of the critical section.](image-url)
Some parts of code executed when updating the critical section require atomic execution to prevent concurrent access. However, to reduce contention on the program atomic parts and to maximize simultaneity of execution of processing engines and GUIs, an approach that minimizes the parts of protected code has been used. For this purpose, the critical section features ready flags. Therefore, either the processing engine or the UI is free to exchange the swap areas when needed. In the presence of unlucky execution timing, it is possible that a swap area is exchanged while it is still being written by the other part of the critical section. However, ready flags provide protection from concurrent access. If the flag is set to false, meaning that the critical section is being written, the information is not read and the update is marked as pending. Then the update will be performed during subsequent critical section updates. Listing 3.6 provides a simplified version of the source code used for updating the critical section. Atomic parts of code are marked with locks to facilitate readability.

Listing 3.6: Simplified version of the source code used for updating the critical section.

```plaintext
Get Current Swap
IF (Change Pending)
  LOCK Changes Lock
  Get Changes Ready Flag
  UNLOCK Changes Lock
  IF (!Changes Ready)
    Skip This Turn
  ELSE
    Read Changes
LOCK Swap Area Lock
IF (!Changed)
  UNLOCK Swap Area Lock
  Skip This Turn
ELSE
  Unset Changed Flag
  Rotate Swap
UNLOCK Swap Area Lock
LOCK Changes Lock
  Get Changes Ready Flag
  UNLOCK Changes Lock
  IF (!Changes Ready)
    Set Pending Changes Flag
    Skip This Turn
```

Instead, Listing 3.7 provides the source code used when writing a value in the critical section. As it may be remarked, atomic sections are used for protecting limited parts of code before and after the write operation. Thanks to ready flags, the value can be written without protection. Therefore, any time-consuming operation required when updating the parameter value may be freely performed without risk of obstructing the other part of the critical section.
Chapter 3. DSPE Base Software Architecture

Listing 3.7: Simplified version of the source code used for writing a value in the critical section

```plaintext
1. LOCK Swap Area Lock
2. Get Current Swap
3. UNLOCK Swap Area Lock
4.
5. LOCK Changes Lock
6. Unset Changes Ready Flag
7. UNLOCK Changes Lock
8.
9. Set Value
10.
11. LOCK Changes Lock
12. Set Changes Ready Flag
13. UNLOCK Changes Lock
14.
15. LOCK Swap Area Lock
16. Set Changed Flag
17. UNLOCK Swap Area Lock
```

This low-cost synchronization approach minimizes thread contention. Furthermore, to additionally reduce the amount of operations performed by the critical section, it is possible to customize the critical section by specifying the policies and rates used for performing the updates. The solution utilized for structuring the swap areas also contributes to the minimization of the critical section overheads. A combination of an array and a linked list is used to favour scalability. Parameter values are written by directly accessing the needed position of the array. When a parameter is first modified, a reference is added to the linked list. When consuming parameter updates during critical section updates, the linked list is traversed for preventing unnecessary updates.

Apart from Runner parameters, it is possible to exchange other information with the critical sections between user interfaces and processing engines. This information comprises

1. The application execution state with the purpose of allowing programatically driving the application's execution. For instance, it is possible to stop or freeze the application from Unit Behaviours.

2. Software component bypasses, which allow disabling and enabling the application software components at run time in the user interface. The infrastructure for component bypass is generated automatically by DSPE.

3. Errors thrown by the application's infrastructure or by algorithms present in Unit Behaviours (see See Section 3.25 for more details).

4. Console/log information and values of the execution profiles of applications, produced at run time by the application's infrastructure or by Unit Behaviours (see See Section 3.25).
3.23 Managers

As a way to favour source code reuse and extension, the software architecture generated by DSPE features managers. These are centralized source code locations featuring special-purpose infrastructures. The functions provided by Managers may be utilized by the generated source code or used by algorithms inside Unit Behaviours and Runner Behaviours. In some situations, these functions simply wrap the C functions provided by standard libraries. This solution has been adopted, in DSPE, to favour source code separation and to ease portability. Table 3.3 provides a list of the available managers.

Table 3.3: List of the managers provided by DSPE.

<table>
<thead>
<tr>
<th>Manager Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory manager</td>
<td>features the functionality of the memory infrastructure for memory allocation, management, and disposal. More information is provided in Section 3.24.</td>
</tr>
<tr>
<td>String manager</td>
<td>is used to handle arrays of characters. For instance, it provides functions for copying strings.</td>
</tr>
<tr>
<td>Engine manager</td>
<td>provides functionalities for modifying the application execution state programmatically.</td>
</tr>
<tr>
<td>Thread manager</td>
<td>features the threading infrastructure. It’s a centralized repository of functions for managing and synchronizing independent threads of execution. More information is provided in Section 3.24.</td>
</tr>
<tr>
<td>Coprocessor manager</td>
<td>provides the infrastructure for parallel processing with soft coprocessors. More information is provided in Section 6.11.</td>
</tr>
<tr>
<td>Platform manager</td>
<td>wraps platform-specific functionalities and provides them uniformly to the rest of the generated application.</td>
</tr>
<tr>
<td>Error manager</td>
<td>features the error infrastructure. Allows collecting errors that may occur during application execution and allows reacting to these errors in a fail-safe way. Section 3.25 provides more information about the DSPE support for errors.</td>
</tr>
<tr>
<td>Info manager</td>
<td>provides the console/log infrastructure for displaying textual information at application run time. It is particularly useful for debugging and fine-tuning. See Section 3.25 for more information.</td>
</tr>
<tr>
<td>Profile manager</td>
<td>features the execution profile support, which allows optimizing applications. See Section 3.25 for more information.</td>
</tr>
</tbody>
</table>

In DSPE, care was taken to provide thread-safe manager functions. Consequently, it is possible to contemporaneously use managers in more than a thread of execution. Furthermore, coherently with real-time stream-processing requirements, the execution time of all manager functions has been limited to very short intervals. Therefore, slow downs and interruptions caused by managers should be unnoticeable.
3.24 Memory and Thread Management Infrastructures

Specific features of the source code of real-time stream-processing applications may require special-purpose infrastructures for memory and thread management. These are, for example, specific functionalities for memory allocation. Access to these functionalities may be useful in the algorithms encapsulated in Unit Behaviours too. As a consequence, in DSPE, these infrastructures have been isolated from the rest of the software architecture and are accessible from every part of the source code. In addition, some features of the standard libraries that are useful for memory and thread management have been wrapped by the DSPE infrastructures to allow intercepting errors and to favour portability.

Concerning the infrastructure for thread management, it deeply integrates with the soft coprocessors and the accelerator infrastructures, used for the parallel processing versions of the stream-processing applications (see Chapter 4, Chapter 6, and Chapter 7 for more details).

3.25 Errors, Console/Log Information, and Execution Profiles

As introduced in Section 3.22, the source code generated by DSPE features debugging and optimization supports, which both ease the development and fine-tuning of applications. The provided functionalities are optional and may be activated at the DSPE model level. There is

- The error infrastructure: allows signalling errors when failures or exceptional situations arise. Pre-defined errors are thrown by the application infrastructure. Furthermore, it is possible to specify custom errors. Depending on the type of error, the application execution state may be automatically modified—for instance, by stopping processing. Critical errors may even force the application into the shutdown phase.

- The console/log infrastructure: allows keeping track of execution information coming from the application's infrastructure or from Unit Behaviours. This information can be displayed on the application's console or stored in a log file. Mainly, this infrastructure is a replacement of standard solutions like printf(), which is not adequate for stream processing as a consequence of its overheads. The DSPE console/log infrastructure is lightweight and does not obstruct the processing engines.

- The execution profile support: allows collection information, such as execution time, throughput, latency, and response jitter, concerning the application, the software components and the infrastructural parts. Furthermore, this support also provides information about the behaviour of schedulers, the event history, and the occupation of the task queues when using the event-driven and the soft coprocessor infrastructures (see Chapter 5 and Chapter 6, respectively). Execution profiles may be displayed on the console or stored in special purpose files, which can be subsequently used by DSPE for performing profile-guided transformations.
Parallel Stream-Processing Approaches

Today, chip-level multiprocessors are present in most personal computers and in a growing number of embedded systems. The available variety and the affordable costs make them ideal for a wide spectrum of deployments, from mobile to high-performance computing. General-purpose processors are typically obtainable as multi-cores, with 4, 6, 8, or so cores. Special-purpose processors, like GPUs, provide, instead, hundreds of cores. Their architecture and features vary significantly. However, all processors share the commonality of being parallel and, consequently, have to be programmed and operated with software solutions that take into account parallelism.

The development of parallel programs is not a simple task. Possibilities of parallel processing are often hidden in the data that need to be processed, in the algorithms used for processing, or in the software architecture of applications. Furthermore, the exploitation of most forms of parallelism is complex and requires adequate support in the development tools and in the execution infrastructure used.

As introduced in Chapter 1, the development of parallel real-time stream-processing applications may be particularly difficult because the processing they perform typically implies computation intensity and time-critical requirements. However, in the context of parallel processing, stream processing also introduces some advantages. Associated with stream processing are some forms of parallelism that may be exploited systematically and that allow performance improvements without excessive waste of precious resources like memory. These forms of parallelism (see Section 4.1 and Section 4.2) take advantage of the nature of stream-processing applications and are effective when used on multi-cores and many-core processors. Utilization of these forms requires both design-level and implementation-level solutions:

1. First, processing performed by the application has to be decomposed into independent parts to allow concurrent execution on the available processor cores. A typical approach is to process more stream items contemporaneously by means of data parallelism. Alternatively, it is possible to execute the stream-processing algorithms in simultaneity
Chapter 4. Parallel Stream-Processing Approaches

through task or data flow parallelism. More information about the parallel processing decompositions supported by DSPE is provided in Section 4.2.

2. On the other hand, a solution for evenly assigning the decomposed parts to the available processor cores has to be found. Two main approaches are available: static compile-time partitioning and dynamic load balancing. With static assignment, low-cost synchronization between the parallel processing parts of the application is possible, but there are risks of load imbalances and poor scalability. With dynamic assignment, the parallel processing infrastructure may introduce more overheads, but better balance and scalability may result.

The most effective solutions are application and hardware dependent. Therefore, DSPE takes advantage of domain-specific development techniques (see Chapter 2) to allow the systematic exploitation of the available parallelism. The DSPE generators automatically adapt both the parallelisation approach and the parallel processing infrastructure to specific requirements. This chapter presents and analyses design-level and implementation-level challenges of parallel stream processing. Furthermore, solutions provided by DSPE are introduced and discussed. This chapter provides the conceptual background on which the parallel processing solutions of DSPE are based. These are discussed in the subsequent chapters, in particular Chapter 6 and Chapter 7.

4.1 Parallelism in Stream-Processing Applications: Granularity

Many forms of parallelism are available in stream-processing applications. Some of them are rather unsophisticated and can be exploited methodically and automatically. Because of that, stream processing solutions prove effective to perform parallel processing on multi-core and many-core processors. In stream-processing applications, possibilities for parallel processing may be present in the data or in algorithms, either at a fine-grained level involving individual stream values and algorithm operations, or at a coarse-grained level involving blocks of values and complete algorithms.

Instruction-level parallelism, used for instance in super-scalar processors for executing more than one instruction during the same clock cycle, is fine-grained and involves individual operations. In stream-processing applications, this form of parallelism is mostly present within the algorithms and is exploited automatically by compilers and processors that are able to overlap the execution of instructions. Therefore, when stream processing, the benefits from this type of parallelism are mostly implicit. Apart from exceptional situations, no particular care should be needed when developing and compiling applications.

Vectorization is, instead, a solution for fine-grained data parallelism that focuses on the single instructions multiple data (SIMD) approach as defined by the Flynn’s Taxonomy (more information is provided in the scholarly literature [96, 57]). This type of parallelism is also frequently present in stream-processing applications at the algorithms level. Vectorization is supported
4.1. Parallelism in Stream-Processing Applications: Granularity

by most general-purpose processors and DSPs. Specific vector processor extensions and specific instruction sets, such as the Intel Streaming SIMD Extension (SSE), are featured by these processors to allow executing the same operation on more values at the same time. To utilize this form of parallelism, most compilers provide auto-vectorization optimizations. Therefore, similarly as for the instruction level parallelism, no particular care should be required when exploiting this form of parallelism in the stream-processing applications generated with DSPE.

With multi-core and many-core processors, other types of parallelism are effective for parallel stream processing:

**Loop-level parallelism** [96] typically focuses on processing fine-grained portions of data with the same set of operations concurrently. It is frequently used in large loops, for instance, by contemporaneously executing different iterations of the same algorithm on adjacent values. For example, OpenMP [107, 23] is a language extension allowing various forms of loop-level parallelism. As highlighted below, this type of parallelism may imply significant synchronization overheads when parallel processing. Consequently, in stream-processing applications, it is mostly used for the application's most time-consuming parts. Loop-level parallelism is frequently exploited on GPUs, by taking advantage of the approach called single program multiple data (SPMD) [96]. GPUs feature a hardware architecture ideal for executing a large number of concurrent threads that process individual values with the same set of operations (see Chapter 7 for more information). This set of parallel operations is specified with GPU-kernels, which are frequently made of the source code contained in loops.

**Coarse-grained parallelism** [55, 89] is the form of parallelism most easily utilizable in stream-processing applications. It is exploited at the software-component level by executing complete algorithms on portions of the data. For instance, in the audio domain, the granularity of a block of data could be 256 or 512 audio samples. In stream-processing applications, coarse-grained parallelism may be present, both at the data and algorithm level. When stream processing, coarse-grained data parallelism is more effective because stream items tend to be of a homogeneous type and are malleable (the size of a block of data can be dynamically adapted); therefore, it is easier to flexibly decompose streams than algorithms.

In stream-processing applications, coarse-grained parallelism is frequently easier to exploit and may provide advantages. Algorithms used for stream processing typically feature high data locality. Coarse-grained parallelism allows profiting from this locality of processing. Concurrency in programs usually implies performance trade-offs caused by synchronization and communication overheads between parallel tasks. For instance, on a multi-core general-purpose processor, an L1-cache miss may cost around 10 cycles, whereas an L2-cache miss may be of 200 cycles. False sharing between caches introduces further performance penalties. If data locality is available, with coarse-grained parallelism it is possible to exploit the processors’ caches efficiently, consequently reducing access to global memory and decreasing the amount of cache incoherencies (for more information about these topics see Section 4.6). Furthermore, locality of processing reduces the amount of required synchronization between
Chapter 4. Parallel Stream-Processing Approaches

independent processing units. Therefore, whenever possible, it is first important to implement concurrency at the highest possible level. Then, fine-grained approaches can be adopted for specific heavyweight parts of the application.

Nonetheless, to profit from the performances provided by the modern multiprocessors, different approaches both at the fine-grained level and coarse-grained level have to be combined. In particular, if heterogeneous hardware made of multi-core and many-core processors is used (see Chapter 7), integration of coarse-grained and fine-grained parallelism typically yield strong speed-ups.

As previously introduced, thanks to component-based development, DSPE allows explicitly specifying some forms of coarse-grained parallelism at the model level. Therefore, generators may use this abstract information to automatically produce the parallel processing infrastructure for these forms of parallelism. The next section provides more information about this subject. DSPE supports coarse-grained data and task parallelism for multi-cores and, by integrating with C for CUDA, fine-grained data parallelism for GPUs (see Chapter 7 for more information). For instance, when using DSPE for a video processing application, coarse-grained parallelism may be used for processing complete images in parallel, whereas fine-grained parallelism may be used at the pixel level. In DSPE, integration with OpenMP and other parallel programming language extensions is also allowed to exploit loop-level parallelism on multi-cores. Furthermore, decompositions at the algorithm level may also be performed by directly using the thread manager (see Section 3.23 and Section 3.24).

4.2 Parallelism in Stream-Processing Applications: Decompositions

In a way similar to all other types of programs, stream-processing applications may be decomposed by either task or data decomposition. Additionally, an alternative that is considered a derivation is provided by the data flow decomposition. See the scholarly literature [57, 123, 96, 55, 89] for more details on these decompositions.

By taking advantage of component-based development and by profiting from the block-processing support (see Section 3.19), the DSPE model allows specifying the following forms of stream-processing decompositions:

Component decomposition is a form of task decomposition that is allowed by specifying independent components working concurrently on the same data. At each execution round, different parts of the application are executed simultaneously on some portions of the streams. With this type of decomposition, components that process in parallel have to be independent from each other. In particular, processing performed on a portion of stream by one component should have no influence on the execution of other components that process the same portion. Furthermore, no particular order of execution should be required.

Stream-flow decomposition is a form of data flow decomposition that is performed by chain-
4.2. Parallelism in Stream-Processing Applications: Decompositions

Processing and simultaneously executing components. Each component processes independent portions of the stream. When the first component executes on new data, the second one executes on the data previously processed by the preceding component and so forth.

**Block decomposition** is a form of data decomposition, which is allowed by letting the same component execute in parallel and simultaneously process adjacent portions of the stream. Block decomposition requires the possibility of independently processing the stream portions. Algorithms have to be stateless: no internal state of the algorithm that is used across subsequent stream portions has to be present (see Section 6.9 for more details). These types of internal states are, for instance, algorithm memories like filter delay lines. If they are used when processing, serial processing is imposed, which prevents the utilization of the block decomposition.

**Sub-block decomposition** is another form of data decomposition available inside software components. If software components are stateless, the stream portions may be split by the components into sub-portions that are concurrently processed. Sub-block decomposition is mostly useful, if the applications contain stateless components with long execution time. It allows locally sub-splitting the data stream to increase concurrency and reducing processing latency. On the other hand, DSPE also supports block aggregations. These are particularly useful if the execution time of the algorithms is short. With block aggregations, the software component combines two or more portions of the stream to improve locality of processing and to reduce the overheads of parallel processing.

**Kernel decomposition** is a form of data decomposition supported by the DSPE accelerator infrastructure (see Chapter 7 for more details). At the DSPE model level, it is possible to specify how the stream portions have to be automatically decomposed into CUDA Grids and CUDA Blocks.

![Figure 4.1: Example of different types of decompositions in a stream-processing application.](image-url)
Blocks to perform parallel processing with GPU kernels.

In DSPE, all these forms of decomposition can be combined in compound solutions. See Figure 4.1 for an example. In general, the forms based on data decomposition, which are block, sub-block and kernel, produce better results. Streams are more malleable than algorithms and can be dynamically subdivided into fairly homogeneous parts. Thereby, with the block, sub-block, and kernel decompositions, it is frequently possible to obtain throughput increases and latency reductions proportional to the number of resources used for parallel processing. Additionally, at model level, data decompositions are easier to specify because they involve only individual software components. Instead, the improvements provided by the task and data flow decompositions are mostly bound to the amount of present parallel processing components and to their execution times. In stream-processing applications, task and data flow parallelism are harder to exploit because algorithms are normally difficult to split in homogeneously weighting parts. In the attempt to allow balancing the execution time of the application software components, the DSPE development environment provides model-to-model transformations, which allows splitting and merging software-components at the model level. See Section 9.8 for more information.

### 4.3 Parallel Design Patterns for Stream Processing

By utilizing the described stream-processing decompositions, in DSPE it is possible to design and implement various parallel design patterns. These patterns are specified at model-level and provide abstract information about the interactions between the software components of the stream-processing applications. As it will be described in Section 6.3, the source code generated by DSPE also utilizes parallel design patterns as part of the generated execution environment. The resulting application combines the model-level patterns and the execution-environment patterns into an aggregate and specific parallel processing solution.

In DSPE, the following model-level patterns are possible:

**Fork-join patterns** [96] are allowed by taking advantage of the component decomposition. With a split component, it is possible to separate a stream into two or more independent flows of data, which may be processed independently by other components and, then, merged by a join component.

**Divide-and-conquer patterns** [96] are allowed by component decompositions. By repeatedly halving the data stream into sub-streams, it is possible to process each obtained half independently and concurrently.

**Scatter-gather patterns** [57] are mostly possible with component decomposition and relate to fork-join and divide-and-conquer patterns. Scatter components split the work to perform into parts that can be processed independently. Then, after processing, gather components collect and combine all the obtained results.
4.4 Performance Improvements from Parallel Stream Processing

Copy-on-write patterns [96] are by default used in all DSPE components. Normally, the component algorithms read information from a memory location and write it in a different place. As explained in Section 3.20, this solution allows more flexibility in terms of possible component connections. The copy-on-write pattern is also very useful in the context of parallel processing, because it reduces the risk of concurrently accessing the same memory regions.

4.4 Performance Improvements from Parallel Stream Processing

Performance improvements, which may result by using the parallel stream-processing infrastructure of DSPE, are of different types. Mostly, depending on the utilized decomposition approach and the used parallel processing hardware, an application speed-up is obtained with consequent increase of the application throughput. If required, this additional throughput allows improving the quantity or quality of processing performed by the application.

Alternatively, in some situations a reduction of latency may also result. This type of improvement is, however, harder to obtain in stream-processing applications because the infrastructure required for parallel processing needs to internally buffer some additional data (as introduced in Section 1.2). Nonetheless, if the latency reduces, it is possible to improve the application's real-time behaviour, for example, by enhancing the application's interactive features. In terms of latency reduction, the data decompositions are the most effective, whereas the different forms of task and flow decompositions mostly contribute in the form of latency hiding: light components are executed behind the bottleneck component. In addition, in parallel stream-processing applications, it is also possible to perform communication and synchronization between components in simultaneity with their execution and, therefore, obtain even more improvements.

A variety of factors influence the attainable results. Mostly, these factors relate to the nature of the application and parallelization approach used. For example, performance may be affected by the amount of control task performed by the application, by the type of the stream items, and by particular properties of the processing algorithms. On the other hand, external factors have great influence too. These are, for instance, characteristics of the hardware and the OS that are used.

The sections that follow provide further details about the performance obstacles in parallel real-time stream-processing applications.

4.5 Data Dependencies and Bottleneck Components

Migrating a stream-processing application for execution on parallel processors may prove difficult. Data dependencies and bottlenecks caused by the algorithms and by the processing infrastructure may limit parallelism.
Figure 4.2 provides an example of the high-level design of a stream-processing application in DSPE. Some of the dependencies existing between algorithms are due to the connections present between software components. These connections, which represent the flow of the stream inside the application, impose some degree of serial processing. Parallel processing is still possible, by concurrently executing the components on different portions of the data; however, performance improvements may be unavoidably limited. The algorithms encapsulated in software components, if state-full, represent further important data dependencies. Delay lines, populating the internal state of IIR digital filters, are an example of the most typical obstacles. To avoid corrupting the internal state of the filter, parallel processing of more portions of the stream with the same filter may be precluded. A stream-processing application may be built of both state-less and state-full software components (see Section 6.9 for more details). With state-less components, it is possible to use block and sub-block decompositions. On the contrary, state-full components permit only component or stream-flow decompositions and, therefore, the potential parallelism may reduce.

Mostly, in parallel stream-processing applications, these design obstacles are critical if they concern the bottleneck component. A component with long execution time may slow down all components that precede or follow it. Preceding components have to stop processing to avoid overloading the bottleneck component, whereas subsequent components, before continuing processing, have to wait for new data from it (see Figure 4.3 for an example). As a consequence, in a complete stream-processing application, all components normally have to execute at the rate of the slowest bottleneck component.

Data dependencies and bottlenecks are very dangerous in stream-processing applications because processing is continuous. In most situations, the same set of components is used for
4.6 Technical Aspects of Shared-Memory Multiprocessors

The source code generated by DSPE mostly provides functionalities that target shared-memory symmetric multiprocessors (SMPs), today widely available as multi-cores in general-purpose and embedded devices. With the addition of the accelerator infrastructure, which is discussed in more detail in Chapter 7, heterogeneous combinations of multi-core CPUs and many-core GPUs are also supported.

As introduced in Section 4.1, multi-core processors utilize caches to limit the costs required for loading the data from main memory to the CPU registers. Each time a core of the processor requires some data, a certain amount of information, which includes the data, is loaded into the caches of the core. This amount of information is called a cache-line. Caches provide faster
access than main memory and, therefore, allow profiting from locality of processing on the cached information. Subsequent operations performed on the same data or on near data in memory will be faster, because frequently the data are already present in the cache-line. Most processors provide a hierarchy of cache levels, which are called L1, L2, etc., each of different size and featuring different access speeds. Furthermore, processors also provide caches for instructions. If correctly exploited, caches allow significant performance improvements.

In parallel programs, the impact of caches is significant because the processor cores may need to share the data contained in the caches, consequently causing cache-misses. A cache miss happens when a core is not able to find the required information in its caches. Cache misses represent costs. An L1 cache miss normally costs around 10 clock cycles, whereas L2 cache miss up to 200 cycles. Every time a core requires the data modified by another core, the cache lines that contain the data have to be copied first to a lower cache level or to global memory and then to the cache of the core that has to process the data. Furthermore, with multi-cores, false sharing has an impact. Cache lines may contain some data, which are not directly required by the core that caches them. If these data are instead required by another core, the cache lines are invalidated because they need to be copied, even if not necessary, with consequent unnecessary overheads in terms of cache misses. Therefore, on multi-cores, if memory is not appropriately managed, important performance penalties may result. As it will be introduced in the next section, DSPE has functionalities that allow profiting from caches in parallel stream-processing applications. An article by Drepper [40], provide more information about the effect of caches on the performance of parallel processing applications.

### 4.7 Technical Aspects of Preemptive Operating Systems

The source code generated by DSPE mostly supports the execution of applications with OSs that perform preemptive scheduling with time slicing. These types of operating systems are the most common. To allow running more than one independent thread of execution concurrently, the OS assigns portions of execution time called to the threads. Each thread with assigned time slices executes on a core as long as the time slices elapse or as long as the thread needs to stop, for instance when waiting for some information. Then, if required, the OS assigns the core to another thread, with a consequent context switch for substituting the thread information present on the core. Context switches represent costs. The assignment of time slices to the threads is done dynamically, depending on the threads’ priorities and depending on the available processor cores.

Other types of scheduling approaches exist. The applications generated with DSPE feature their own event-driven internal scheduler and delegate scheduling to the OS only for the execution of the soft coprocessors (see Chapter 6). For this reason, the execution of the DSPE-generated code does not strictly depend on the OS scheduler. However, the current version of DSPE has not been tuned and tested for use with other types of OS scheduling approaches; therefore, in this and other chapters, discussion concentrates on the generated code behaviour.
4.8 Stream Decomposition: Grain Size

The amount of data processed during each processing cycle has an important influence on the parallel processing behaviour of stream-processing applications. If small data amounts are processed, the application may spend excessive time scheduling and synchronizing software components, whereas, with a reasonable data amount, locality of processing may improve. As introduced in Section 4.1, in parallel applications, more locality of processing means less synchronization overheads and, because of fewer cache misses and false sharing, better utilization of the processor caches.

In DSPE, it is possible to profit from the block processing support (see Section 3.19) for configuring the grain size of the stream decomposition and, consequently, tuning the application's parallel processing behaviour. Normally, streams may be easily separated into homogeneously sized parts. Furthermore, it is also normally possible to modify the block size, even dynamically at execution time, according to different application and hardware needs. For this purpose, DSPE also features sub-block decompositions and aggregations (see Section 4.2), which allow adapting the grain size of the streams with the purpose of enhancing parallel processing. The stream item size and the stream value density may significantly influence the selected grain sizes. For example, very different sizes may be used for a stream of images than for a stream of audio samples. In stream-processing applications, well-tuned grain sizes are of advantage to all types of parallel decompositions. However, software components featuring data decompositions may have particular benefits. As previously introduced, compared with decompositions executed at the algorithm level, in stream-processing applications, decompositions at the data level are easier to perform and provide more proficient results.

When tuning the block size of a parallel stream-processing application, care is needed. Attempts to simply use large block sizes are not sufficient. If the block size is excessively large,
parallel processing may suffer. For example, with a large block size, the required amount of data may saturate the caches, and some of the advantages of the data locality may be lost. Similarly, large block sizes reduce the fluidity of the data stream. Efficient forms of interleaving of the algorithms on the processor cores may be precluded, with consequent wait states and load imbalances. Additionally, large blocks significantly increase the processing latency, which needs to be constrained in real-time applications.

4.9 Run-Time Interferences, Execution Pressure, and Load Imbalances

As introduced in Section 4.5, data dependencies and bottleneck components may represent performance limits in parallel stream-processing applications. At run time, their negative impact may be even more substantial. For example, if the user interacts with the application while processing, the components’ weights may change dynamically; therefore, the influence of data dependencies and bottlenecks on the speed-ups from parallel processing may become more significant.

In terms of performances and real-time behaviour, any type of alteration of the execution time of the algorithms and other parts of the application represents a serious danger when parallel stream processing. On chip-level multiprocessors, the execution time of the algorithms may be altered by many run-time interfering factors. These interferences may be present in the form of interruptions or slowdowns and may be caused, for instance, by

- Thread context switches performed by the operating system's preemptive scheduler. These are particularly annoying in all situations of over-threading (when there are more software threads than available hardware cores).

- Thread context switches caused by synchronization (locks) that slow down processing or prevent algorithms from completely consuming their assigned time slices.

- Thread priorities and thread affinities that reduce the amount of time slices assigned to certain algorithms or unevenly assign the available cores to the processing threads.

- Memory access latencies. As introduced in Section 4.6, cache misses represent costs, which may vary, depending on the accessed cache level. False sharing increases the impact of these costs.

- Virtual memory swapping caused by the memory management system, if the available amount of RAM is insufficient.

- File or network access that causes long wait states, in particular if shared resources are accessed concurrently.
### 4.9. Run-Time Interferences, Execution Pressure, and Load Imbalances

Table 4.1: Example of execution-time variations caused by run-time interferences. Minimum (min), mean, and maximum (max) execution times are reported in milliseconds (ms) for four application executions.

<table>
<thead>
<tr>
<th>Filter name</th>
<th>Serial version</th>
<th>mean (ms)</th>
<th>max (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>CENTER WIDE</td>
<td>0.005585</td>
<td>0.005699</td>
<td>0.015708</td>
</tr>
<tr>
<td>HRTF FIR Left Left</td>
<td>0.330921</td>
<td>0.337229</td>
<td>0.429010</td>
</tr>
<tr>
<td>HRTF FIR Right Left</td>
<td>0.330921</td>
<td>0.336880</td>
<td>0.676153</td>
</tr>
<tr>
<td>HRTF FIR Right Right</td>
<td>0.330921</td>
<td>0.336600</td>
<td>0.430057</td>
</tr>
<tr>
<td>LIMITER</td>
<td>0.002094</td>
<td>0.002281</td>
<td>0.012567</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Filter name</th>
<th>2-core version</th>
<th>mean (ms)</th>
<th>max (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>CENTER WIDE</td>
<td>0.005585</td>
<td>0.006361</td>
<td>0.356752</td>
</tr>
<tr>
<td>HRTF FIR Left Left</td>
<td>0.330921</td>
<td>0.339562</td>
<td>0.932721</td>
</tr>
<tr>
<td>HRTF FIR Right Left</td>
<td>0.330921</td>
<td>0.339405</td>
<td>1.569080</td>
</tr>
<tr>
<td>HRTF FIR Right Right</td>
<td>0.330921</td>
<td>0.339573</td>
<td>1.590373</td>
</tr>
<tr>
<td>LIMITER</td>
<td>0.002094</td>
<td>0.003410</td>
<td>0.216076</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Filter name</th>
<th>12-core version</th>
<th>mean (ms)</th>
<th>max (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>CENTER WIDE</td>
<td>0.005934</td>
<td>0.006760</td>
<td>0.032464</td>
</tr>
<tr>
<td>HRTF FIR Left Left</td>
<td>0.344883</td>
<td>0.365347</td>
<td>0.746317</td>
</tr>
<tr>
<td>HRTF FIR Right Left</td>
<td>0.344883</td>
<td>0.364799</td>
<td>0.751553</td>
</tr>
<tr>
<td>HRTF FIR Right Right</td>
<td>0.344883</td>
<td>0.365911</td>
<td>31.601167</td>
</tr>
<tr>
<td>LIMITER</td>
<td>0.002444</td>
<td>0.003410</td>
<td>0.216076</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Filter name</th>
<th>24-virtual-core version</th>
<th>mean (ms)</th>
<th>max (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>CENTER WIDE</td>
<td>0.006981</td>
<td>0.009631</td>
<td>0.128110</td>
</tr>
<tr>
<td>HRTF FIR Left Left</td>
<td>0.358497</td>
<td>0.540103</td>
<td>47.396339</td>
</tr>
<tr>
<td>HRTF FIR Right Left</td>
<td>0.357450</td>
<td>0.532540</td>
<td>47.291618</td>
</tr>
<tr>
<td>HRTF FIR Right Right</td>
<td>0.359195</td>
<td>0.537181</td>
<td>47.419029</td>
</tr>
<tr>
<td>LIMITER</td>
<td>0.002793</td>
<td>0.005056</td>
<td>0.017105</td>
</tr>
</tbody>
</table>
Chapter 4. Parallel Stream-Processing Approaches

As a consequence, the parallel processing behaviour of the application may become unpredictable. For example, Table 4.1 provides measurements of the execution time of some filters, part of a stream-processing application executed both as mono-core version and multi-core versions. These measurements were obtained on a computer featuring 12 physical cores and 24 virtual cores with hyper-threading. The interferences present in the example of Table 4.1 are caused by memory access latencies and context switches. The measurements highlight that the interferences are stronger for application executions with a large number of cores. Hyper-threading also has an impact.

During the complete execution run, these variations represent a danger in terms of parallel stream processing. Instabilities may have a strong impact on performances, if they compromise the application execution pressure. In a stream-processing application, a variation in the weight of an algorithm may produce reactions and side effects in the execution of algorithms that precede or follow it in the processing chain. As a consequence, software components may need to wait to avoid overloading other components or may become idle for momentary lack of samples to process. If the application loses its internal execution pressure, load imbalances on the available processor cores may appear, which reduce the positive impact of parallelism, consequently compromising the application performances (Section 6.14 and Section 6.16 provide more information about the execution pressure in parallel stream-processing applications). In real-time stream-processing applications, it may be hard to obtain continuously balanced execution loads with constantly full utilized processor cores. Load imbalances strongly reduce or even annihilate the speed-up from parallel processing. For this reason, the infrastructure provided by DSPE focuses on dynamic load balancing solutions, which allow adapting the execution of software components and the assignment of processing tasks to the cores at run time. More details about this topic are provided in Section 4.12 and in Chapter 6.

Furthermore, if heterogeneous processors with distributed memories are used, further challenges have to be faced. The data of the streams may need to be copied between the available memories with consequent additional costs and run-time interferences. Furthermore, heterogeneous systems face the problem of the bottleneck processor. In a way similar to bottleneck components, overloaded processors may slow down other processors and, consequently, limit throughput and scalability. DSPE provides the accelerator infrastructure with the purpose of facilitating the execution of stream-processing application on heterogeneous hardware made of multi-core CPUs and GPUs. As a solution to the problem of the bottleneck processor, the accelerator infrastructure provides heterogeneous work stealing (see Section 7.9 for more details). More information about the challenges represented by heterogeneous processors are provided in Chapter 7).
4.10 Non-Blocking Synchronisation: Spinlocks and Atomic Actions

Most operating systems for SMPs provide infrastructure for executing parallel programs by utilizing independent threads of execution that run concurrently. All these threads have contemporaneous access to the same memory region. Their execution may need to be synchronized because there are types of concurrent memory access that need to be performed atomically to avoid race conditions. There are many solutions that allow synchronizing threads.

For example, DSPE uses the multi-threading libraries PThreads and WinThreads, which feature mutexes, semaphores, and other blocking locks that suspend the execution of the running thread, as long as the lock is held by another thread. If the lock is freed, all blocked threads will resume as soon as the operating system scheduler has an available scheduling time slice for their execution. These kinds of locks are valuable when protecting critical regions of code that need to execute atomically. Furthermore, their blocking behaviour increases the possibilities that the lock is freed soon. By blocking, these types of locks free subsequent time slices, and the thread owning the lock may have the chance to resume soon. Therefore, these solutions prove effective when performing multi-tasking on a single core.

However, blocking approaches are not so efficient on shared-memory multi-cores. Normally, threads execute in parallel and each thread has an own assigned core. Consequently, blocking at the lock may be counterproductive. After locking, the core may sit idle. Sleeping threads and context switches degrade the parallel processing performances on multi-cores. Therefore, on multi-cores, threads should block as little as possible. As an alternative, threads may continue processing and performing additional work, or, if this is not possible, spin at the lock and immediately resume processing when the lock is freed. This approach prevents the thread from losing the assigned scheduling time slices.

Therefore, non-blocking synchronization solutions are mostly needed when parallel processing on SMPs. These are implemented, for instance, with spinlocks [4, 98, 144, 122, 114] and atomic operations [38, 99, 123, 52, 67, 5], used extensively in specific parts of the generated infrastructure of DSPE. These types of solutions allow minimizing the synchronization overheads and consequently maximizing performances. In many situations, utilization of atomic operations represent the most efficient solution. However, they allow protecting only small regions of code and are frequently not as flexible as required. Spinlocks are easier to use and provide a more flexible solution.

In DSPE, spinlocks and atomic operations are used in the core parts of the infrastructure for parallel processing (see Section 6.4 and Section 6.11 for more details). However, to minimize the costs of thread synchronization, the DSPE generated code mostly profits from the nature of discrete events. This topic will be introduced in the next section.
4.11 Lightweight Scheduling and Synchronization Infrastructure

SMPs allow performing parallel stream processing in a fraction of the time required for serial processing. However, as introduced in the previous sections, depending on the nature of the streams and algorithms, finding an efficient solution may be a complex task. In particular, stream-processing applications are frequently time critical. Therefore, care is needed to maximize the application throughput while keeping its latency constrained.

To maximize performances, side by side with an effective decomposition approach (as introduced in Section 4.2), a reactive run-time infrastructure is also needed. In particular, the used parallel processing infrastructure should allow executing every operation as an immediate reaction to a request, without implying unnecessary wait-states. Furthermore, in real-time stream-processing applications, most operations need to execute as soon as possible: control operations should execute soon to favour the application interactions, and processing operations need to execute soon to keep the application's latency minimal. As previously introduced, non-blocking synchronization mechanisms are of help, whereas blocking approaches, such as polling, are instead unprofitable.

In this context, the use of discrete events to schedule the software components of the application brings advantages. With event-driven scheduling, control and processing operations are executed as a direct consequence of other operations. There's no global, fixed execution sequence. Scheduled for processing are only the software components that have something to do. Consequently, delays are minimized. Furthermore, events allow scheduling the software components with different priorities. For instance, components performing control operations may be scheduled with higher priority than components performing processing operations, with consequent benefits in terms of real-time behaviour of the application. Chapter 5 provides more information about how DSPE uses discrete events for event-driven scheduling.

On the other hand, it is possible to use events as a form of lightweight synchronization mechanism between concurrently executing components. After they have been prepared, events can be transformed into immutable objects. Typically, events are constructed and filled by a single software component. Then, they are sent to other components, which only read the contained information. Therefore, after preparation, the information contained in events can be made read-only. Read-only information does not require the use of mechanisms for exclusive access, such as locks and atomic operations. As a consequence, the utilization of events for exchanging information between concurrently executing software components is very effective. Care is required only to avoid concurrent access to events while they are being prepared. Furthermore, adequate synchronization is required when exchanging the events between the components. Instead, when the event's information is accessed, there's no need for synchronization. Therefore, costs may be minimized.
4.12. Scheduling Approaches for Parallel Stream Processing

To summarize, in terms of parallel processing, discrete events are advantageous because they feature a great degree of asynchrony. Therefore, in DSPE, events are used with the following main goals:

- To support dynamic scheduling of components. With event-driven scheduling, the flow of events adapts autonomously to potential run-time modifications or momentary instabilities of the computational load. Nonetheless, it is not completely left alone. By means of priorities and special-purpose events, the scheduling of components can be explicitly driven for fulfilling particular needs.

- To provide a lightweight communication mechanism between software components. After they have been sent, events may become immutable and the associated data read-only. Therefore, there is no need to guard against concurrent access of the information inside events.

Chapter 5 and Chapter 6 provide more detail about the support provided for discrete events in DSPE.

4.12 Scheduling Approaches for Parallel Stream Processing

Another important factor that influences the performance of parallel stream-processing applications is how the parallel partitions, obtained with the decomposition approaches used, are assigned to the available processor cores.

There are two main approaches utilized in the available software solutions and documented in the literature [132, 70, 56, 131, 84, 137, 136, 109, 22, 21, 61, 27, 72, 28, 119, 11, 93, 66]. The static approach concentrates on finding an assignment of the parallel processing partitions to the cores at compile time. The dynamic approach focus, instead, on dynamically and autonomously adapting the assignment at run time. Static assignment is advantageous because the required infrastructure can be minimized to improve performance. However, the static assignment approach may suffer from the presence of run-time execution-load variations (see Section 4.9), which may cause load imbalances and consequent non-ideal scalability. Furthermore, in stream-processing applications, the static assignment may be particularly complex to perform in combinations with task and data flow decompositions. As introduced in Section 4.2 and Section 4.8, it is harder to find an homogeneous partition of the algorithms than it is to find an homogeneous partition of the data streams. As a consequence, occupying the cores equally with tasks or data flow decompositions may require significant efforts. On the other hand, the dynamic assignment approach features better balance to most of the real-time stream-processing applications but incurs more overheads in terms of required infrastructure.

By profiting from model-driven code generation (see Section 2.2 for more details), DSPE provides an approach that combines the advantages of both the static assignment and dy-
Chapter 4. Parallel Stream-Processing Approaches

Dynamic assignment approaches. Similarly to the static assignment approach, at model level, the parallel partitions obtained by the decomposition approach used can be optimized before generation time, by utilizing profile-guided model transformations. Then, a run-time infrastructure for parallel processing based on the dynamic assignment approach can be configured and generated specifically for each application and execution environment. The implied infrastructural overheads can, therefore, be minimized. At run time, the generated infrastructure, which is based on soft coprocessors, adapts the assignment of the application parallel partitions to the cores autonomously and, therefore, maximizes the cores’ utilization (Chapter 6 provides the details about the soft coprocessor infrastructure generated by DSPE).

As a consequence, in DSPE, the following factors influence the scheduling of the application software components on the cores:

1. The decomposition approach used for separating the parallel processing software components. As explained in Section 4.2, there are many possibilities. In particular, depending on their execution weight, the software components may be executed independently or in combination with the event-driven scheduler of the application.

2. The behaviour of the event-driven scheduler that is used internally by the application. As explained in Section 4.11, on the basis of priorities and special-purpose events, the event-driven infrastructure of DSPE allows adapting the execution sequence of the components dynamically. For instance, event-driven scheduling may be exploited for maximizing the internal application execution pressure (see Section 6.16 for more details).

3. The scheduling approach utilized by the soft coprocessors. As it will be explained in Section 6.12, the soft coprocessor infrastructure of DSPE can be configured to assign the application parallel partitions with different approaches, which may be based on priorities and on the weight of the parallel partitions.

4. The assignment of the threads of the soft coprocessors to the available cores performed by the OS. As introduced in Section 4.7, this assignment may vary on the basis of the used type of OS, the assigned thread priorities and depending on the amount of available cores.
5 Event-Driven Infrastructure

The event-driven infrastructure featured by DSPE mostly provides dedicated functionality for parallel processing in stream-processing applications. The soft coprocessors and the accelerator infrastructures of DSPE, which will be described in Chapter 6 and Chapter 7, deeply integrate and cooperate with the event-driven support of DSPE. The reason is that DSPE features an approach to parallelism based on discrete events.

Nonetheless, in DSPE, some of the adopted methods and technical solutions for event-driven processing are generic. Therefore, the DSPE event-driven infrastructure could prove useful for non-stream-processing applications too. For instance, in the domain of industrial automation, DSPE could be used for developing pure control applications. Similarly, the development of stream-processing applications for mono-processors is also possible. These types of applications are particularly meaningful for scientific simulations based on discrete events (see the next section for details), or in contexts where the internal processing rate of the application frequently suffers dynamic variations.

However, as introduced in Section 4.11, the DSPE event-driven infrastructure is mostly effective for parallel stream processing because discrete events allow

- **Event-driven scheduling**, which is used for dynamically adjusting the execution sequence of parallel processing software components.
- **Event-based communication**, which is used for lightweight synchronization of concurrently processing software components thanks to the immutable nature of events.

Event-driven scheduling and event-based communication are the most important properties of discrete events that are exploited by the DSPE source code generators. Exploitation of these properties is supported by independent functionalities in DSPE. For instance, it is possible to utilize event-based communication even in Configurations, where event-driven scheduling is not present. Similarly, if there is no need to exchange information in an event-driven way, it is possible to use events only for scheduling.
Chapter 5. Event-Driven Infrastructure

This chapter introduces the event-driven infrastructure of DSPE. Particular attention is dedicated to functionalities specific for parallel stream processing. This chapter establishes the preconditions to the chapters that follow, which concern the soft coprocessor and accelerator infrastructures.

5.1 Discrete Events and Message Passing

In computer science, there are many different uses of the concept of *event*. For example, technical solutions based on events are provided by GUI frameworks, such as QT [115] or WxWidgets [145], for consistently updating the different GUI parts. If the state of a GUI component is modified, the component informs other GUI components by sending update events. The programming languages Java and C# also feature reusable infrastructures based on events and event handlers that are useful for application development. Similarly, solutions based on the notion of event are provided by multi-threading and inter-process communication infrastructures. For example, the WinThreads infrastructure features events for thread synchronization.

Another important utilization of the notion of event is done in the technique known as message passing, frequently used in distributed parallel systems. For example, messages are exchanged by independent processes executing on different computers by utilizing MPI. In this context, a message may be seen as a complex form of event that also delivers structured data or parts of a program.

Furthermore, the concept of event is also used in discrete-event systems present in many application domains, such as industrial automation, communications, networking, and manufacturing engineering. In this context, instead of being used with the main purpose of a communication mechanism, events are mostly utilized for managing transitions from application states to other application states. For example, an important use of discrete events is done in discrete-event simulations [37, 147, 143]. There are also many models and patterns that are based on the notion of discrete event, some of which have been described in Section 1.3. Similarly, other utilizations of the concept of event may be found in UML diagrams and in software architectures for enterprise applications.

As introduced in Section 4.11, events are used in the DSPE-generated code, both as a lightweight synchronization mechanism between threads executing on shared-memory multi-core processors and as a scheduling solution for determining the flow of the application execution.
5.2 Abstract Types of the Event-Driven Infrastructure

In a way similar to the abstract types provided by the base infrastructure (see Section 3.6), DSPE also generates abstract parent C structs for the event-driven infrastructure. These C structs are the root of the event-based hierarchy of DSPE elements. All abstract types of the event-driven infrastructure are extended in the generated source code and provide common functionality useful, for instance, for polymorphic use of the generated DSPE elements. Figure 5.1 provides an overview of the abstract types for the DSPE elements provided by the event-driven infrastructure.

The most important DSPE elements featuring support for events are contained in Table 5.1.

| Events | (abstract type DSPEEvent) are used for exchanging information between software components and for scheduling their execution. Different from the shared-memory regions used by standard Gates, events are prepared by source components and sent, through Event Gates, to target components that consume the event. Therefore, each event is always exclusively owned, either by the source or the target component. |
| Event Units | (abstract type DSPEEventUnit) are Units that are able to receive and send events and, therefore, have at least an Event Gate. |
| Queue Units | (abstract type DSPEQueueUnit) are specializations of Event Units able to receive events and store them in queues for later processing. Queue Units consume events when scheduled for processing. |
| State Unit Behaviours | (abstract type DSPEStateUnitBehaviour) are Unit Behaviours that allow developing finite-state machines specific for stream processing. There are two types of State Unit Behaviours: State Implementations and State Block Optimizations. |
| Schedulers | (abstract type DSPEScheduler) are Structures that execute the contained software components with an event-driven approach: the order of execution reflects the order events that are fired by software components. |
| Event Composites | (abstract type DSPEEventComposite) are Composites able to send or receive events. If a Composite has at least an Event Gate, it is an Event Composite. |
| Event Runners | (abstract type DSPEEventRunner) are Runners able to send or receive events through the critical section. Event Runners allow, for instance, the use of events in the GUI. |
5.3 Events and Event Gates

In the generated source code, Events are C structs that directly contain the data exchanged between Units or, for some specific type of Event, a reference to the memory region containing the data. When Events are sent, instead of copying the complete Event structs from the source Unit to the target Unit, only pointers to the structs are exchanged. This approach favours transparent flow of streams through software components and minimizes the communication costs associated with Events. Units and the contained Unit Behaviours manage the information contained in Events in a way similar to non event-driven applications. As for standard Gates, the information contained in Events is still kept in a common location, but the writer and the reader are not allowed to access it simultaneously. First, a source Unit prepares an Event. Then, a pointer to the Event is exchanged between the source and target Unit. Finally, the target Unit uses the pointer to access the contained information, which for the target Unit has to be considered read-only because of the immutability of events (see Section 4.11 for more details).

DSPE provides different types of Events, ranging from Signals—which are Events without a value—to Message Events, which contain pointers to memory regions allocated dynamically. For each type of Event, there is a type of Event Gate. It is possible to use Events both for Parameter Gates and Data Gates (see Section 3.10). Also available with Events is block processing (see Section 3.19). A single Event is sent for each block of data. Similarly, groups of Events (see Section 3.9) are provided by means of Group Event Gates. If not differently specified, a single Event is also sent for the entire group. Section 8.8 provides more information about the types of Event Gates provided by DSPE.
In the generated source code, Events are exchanged directly between Units (see Figure 5.2). Each Queue Unit owns an event input queue, where Events are queued by other Units and wait their turn for being processed. When a Unit sends an Event, the Event is appended at the end of the event queues of all connected Units. Then, the event-driven scheduler is responsible for scheduling the processing of Units in the order Events have been fired and depending from the assigned priorities. Each Unit with at least an Event in the queue is scheduled for processing. At each turn, the Unit consumes an Event in the queue.

Event queues are implemented as dynamically linked lists to avoid out-of-buffer errors that could potentially occur when using fixed-size arrays.

The C structs generated for Events extend the parent abstract type DSPEEvent, provided in Listing 5.1. The abstract type DSPEEvent mostly features infrastructure for memory management of Events in event pools, as explained in the next chapter.

Listing 5.1: C struct of the DSPEEvent abstract type.

```c
struct DSPEEvent {
    int refCount;
    int blockSize;
    //
    DSPEvent *next;
    DSPEventsPool *pool;
    //
    DSPEvent* (*clone) (DSPEvent *event);
    void (*dispose) (DSPEvent *event);
};
```
5.4 Memory Management: Event Pools

Event pools are used to manage the memory regions of events as follows: if the pool is empty, new events are created, by means of dynamic memory allocation. A reference counter is used in each event for keeping track of the number of DSPE elements referencing the event (see Figure 5.3). As soon as the counter reaches 0, meaning that no more reference is available, the event is appended to the pool for recycling. Therefore, after utilization, events are present in the pool and, as a consequence, reutilized instead of being newly created. In properly functioning stream-processing applications, when sufficient events have been allocated, a reusable event will always be available in the pool. Therefore, dynamic allocation and disposal of memory, which is costly, may be completely avoided.

In DSPE, there is an event pool for each type of Event Gate. Because different block sizes and different kinds of Group Gates may be simultaneously used for each type of Gate, event pools are structured on three levels, which allow storing all variants of events to recycle. The following three dimensions are used for the dynamic structure of the event pools:

- On the first dimension, events are stored depending on their event type.
- On the second dimension, they are stored depending on the kind of Group Gate.
- On the third dimension, depending on their block size.

Special purpose events like initialization events (see Section 5.8) and task-ready events (see Section 6.4) are managed separately with special purpose infrastructures.

Figure 5.3: Illustration of the functionality provided by event pools. A reference counter (rc) is used for keeping track of the number of elements referencing the event.
Like event queues, event pools are implemented as dynamically linked lists to avoid out-of-buffer errors. To limit the memory occupation of event pools, it is possible to define a maximum pool size. Consequently, if an event pool contains enough events, an event with a reference count of 0 is disposed instead of being recycled. Subsequently, it is recreated only if required. In addition, there is support for preconfiguring event pools at initialization time with a default amount of contained events. For more information about the memory management infrastructure provided by DSPE for events, see Section 6.5.

5.5 Event Units and Queue Units

In DSPE, there are five types of software components able to send and receive events: Event Units, Queue Units, Coprocessor Units, Event Composites, and Event Runners. This chapter discusses Event Units and Queue Units. Coprocessor Units are presented in Section 6.2, Event Composites in Section 5.12, and Event Runners in Section 5.13.

As soon as a Software Unit has an Event Gate, either of type input or output, in the generated source code it is converted to an Event Unit. If it has only output Event Gates, it remains an Event Unit. Instead, if it has any input Event Gates, it further specializes to a Queue Unit. It is converted to a Coprocessor Unit if it features soft coprocessor support (see Chapter 6 for more details).

The abstract C struct of Event Units features functions for sending and receiving events. The sendEvent() function is called by Units to dispatch events to connected Units (see Figure 5.4). The queueEvent() function is instead called by the sendEvent() function of connected Units or by the scheduler when events are received.

The code of the base C struct extended by all Event Units is provided in Listing 5.2. In addition to the sendEvent() and the queueEvent() function, Event Units also feature armEvent() and postEvent() functions for preparing and firing output events.

Listing 5.2: C struct of the DSPEEventUnit abstract type.

```c
struct DSPEEventUnit {
  DSPEUnit parent;
  void (*queueEvent) (DSPEEventsUnit *unit, DSPEvent *event, int ID);
  void (*sendEvent) (DSPEEventsUnit *unit, DSPEvent *event, int ID);
  void (*armEvent) (DSPEEventsUnit *unit, int ID);
  void (*postEvent) (DSPEEventsUnit *unit, int ID);
};
```

Queue Units extend Event Units and introduce additional functions for consuming and transiting events (see Listing 5.3).
Figure 5.4: Illustration of the solution adopted in `sendEvent()` functions for dispatching events to connected Units. E1 and E2 are events.

Listing 5.3: C-struct of the DSPEQueueUnit abstract type.

```c
struct DSPEQueueUnit {
    DSPEEventsUnit parent;
    int (*isEventAvailable) (DSPEQueueUnit *unit);
    unsigned int (*getEventID) (DSPEQueueUnit *unit);
    DSPEEvent* (*getEvent) (DSPEQueueUnit *unit);
    void (*transitEvent) (DSPEQueueUnit *unit);
    void (*getTransitEvent) (DSPEQueueUnit *unit, int ID);
    void (*dismissEvent) (DSPEQueueUnit *unit, int ID);
};
```

In DSPE, `sendEvent()` and `queueEvent()` functions are considered public component functions because they are used from the outside of Units for connecting Units with one another. `sendEvent()` functions are installed on Units by Schedulers and Composites. As a technical solution, generators take advantage of the mechanism for dynamic dispatching from external program parts, which is described in Section 3.4. All `sendEvent()` functions are generated as part of a Scheduler, Coprocessor Scheduler or a Composite. Therefore, these functions are not direct members of the Unit that uses them. Functions are installed by using function pointers, thereby transferring the functions to the Unit at run time.

Listing 5.4 provides simplified sample code for a `sendEvent()` function. These functions work as follows: for each Event Gate in a Unit, there is a unique numeric identifier. Depending on the value of the identifier used as an argument when calling the `sendEvent()` function, the related `queueEvent()` function is executed to deliver the event to the connected Unit (see Figure 5.4). Afterward, the source code of the `sendEvent()` function schedules the Unit for processing. `sendEvent()` functions are generated from the abstract information provided in
5.6 Support for Events in Unit Behaviours

the DSPE model. Each time there is at least one connection of type event between two DSPE components, an associated sendEvent() function is produced.

Listing 5.4: Example of the source code for a sendEvent() function.

```c
void sendAnEvent(DSPEEventsUnit *unit, DSPEEvent *evt, int ID) {
    ACoprocSched *sched = (ACoprocSched*) unit->container;
    switch (ID) {
    case FIRST_OUT_EVT:
        Unit1_queueEvent(&sched->Unit1, evt, UNIT1_IN_EVT);
        schedule(sched, &sched->Unit1);
        break;
    case SECOND_OUT_EVT:
        Unit2_queueEvent((&sched->Unit2, evt, UNIT2_IN_EVT);
        schedule(sched, & sched->Unit2);
        break;
    }
}
```

queueEvent() functions are instead auto installed by Units. The function implementation may vary depending on the Unit’s internal architecture. As a default behaviour, queueEvent() functions simply add the received event to the queue of input events waiting for consumption. Consumption of events may happen as soon as the Unit is scheduled for processing.

If an Event Unit has only output Event Gates, no queueEvent() function is installed on the Unit and the function pointer remains unassigned. Similarly, if an event has only input Event Gates, no sendEvent() function is installed. As a consequence, there are generated source code parts that need to test if the function pointer is assigned before calling it. In the case of test failure, the source code reacts with default behaviour or by returning an error.

5.6 Support for Events in Unit Behaviours

To fire events and to access the event input queues from Unit Behaviours inside Event Units and Queue Units (see Figure 5.5), postEvent(), isEventAvailable(), getEventID() and getEvent() functions are provided. Different from sendEvent() and queueEvent() functions, these functions are considered private with respect to the Unit because they are used only internally. Functionality provided by these functions is straightforward. The postEvent() functions allow firing events. The isEventAvailable() function allows testing if there are events present in the event input queue. With getEventID(), it is possible to discover the unique identifier of the Event Gate on which the first event in the queue has been received. The getEvent() function allows accessing this first event.

To access the memory regions, which are associated with events and which are used for storing parameter and data information, shortcuts are provided. These shortcuts are implemented with define directives and behave in a way similar to the shortcuts provided for simple types.
of Unit Behaviours (see Section 3.16 for more information). Shortcuts allow isolating the algorithms from the rest of the software architecture. Furthermore, the generated source code may be configured to automatically load these shortcuts. See Section 5.10 for more details.

5.7 Advanced Event Support: Transit-Dismiss and Arm-Post

Two types of event supports are available: immediate and advanced. Immediate event support is available for all types of Unit Behaviours. Additionally, State Unit Behaviours (see Section 5.9), Coprocessor Unit Behaviours (see Section 6.8), GP Unit Behaviours (see Section 7.4), and Wrap Unit Behaviours (see Section 3.18) provide the advanced version.

The immediate event support provides the functions for retrieving and firing events described in the previous chapter. The featured infrastructure is mostly sufficient for implementing simple Event Units and for utilizing events within Configurations. An event has to be consumed, each time the process function of the Unit Behaviour is called. To allow consuming events inside Configurations, it is possible to configure the generated source code for calling the process function a number of times equivalent to the number of events present in the input event queue. Furthermore, with the immediate event support, output events are prepared automatically by the generated source code. After an event has been sent, a new one is created or recycled. The Unit Behaviour has direct access to the information inside events through shortcuts.

Instead, the advanced support additionally features functions for transiting, dismissing, and arming events. These functions are mostly useful inside schedulers when stream processing because they allow suspending events for a certain amount of time. Coprocessor Unit
Behaviours and GP Unit Behaviours extensively use these functionalities for the purpose of parallel stream processing (see Section 6.10 for more details). All functions for transiting, dismissing, and arming events are also considered private with respect to the Unit. The arm-post support is part of Event Units, whereas the transit-dismiss support is provided by Queue Units.

The transit-dismiss infrastructure provides transit queues, as shown in Figure 5.6. These queues are buffers at the input of units, where events may wait if their immediate release is not possible. There is an independent transit queue for each Event Gate in a Queue Unit. The transit queues' functionality is mainly used for synchronizing events. This type of functionality is frequently required in stream-processing applications. To perform processing, Queue Units may need to use more than one event from more than one of its input Event Gates contemporaneously. Transit queues allow temporarily suspending events, until all the needed events are available.

The arm-post infrastructure provides, instead, buffers of events at the output of Event Units (see Figure 5.6). There is an independent arm queue for each Event Gate of a Queue Unit. The armEvent() functions allow allocating and initializing any number of events, which will wait in queues until they are released with the postEvent() functions. Before postEvent() functions are called, Unit Behaviours may fill events with the information they have to deliver. Afterward, postEvent() functions simply remove the first event waiting in the queue and send it by using the sendEvent() functions.

5.8 Schedulers

Schedulers are the playmakers of the event-driven infrastructure. In the generated source code, Schedulers track the flow of events and execute Units according to the order events are fired or on the basis of priorities.

This approach is different from the scheduling approach used in Configurations (see Sec-
Chapter 5. Event-Driven Infrastructure

The source code generated for Configurations executes Units in round-robin with a fixed execution sequence explicitly specified in the DSPE model. Instead, the source code generated for Schedulers performs event-driven scheduling: each time an event is fired, the receiving Units are scheduled for processing. Schedulers store and manage the sequence of scheduled Units in a queue of Unit references. The default scheduling policy is first-come, first-serve without priorities, but it can be augmented to fit other needs. In particular, support is provided for scheduling special-purpose events, such as task-ready events (see Section 6.4 for more information) with higher priority.

With the purpose of scheduling the first Unit that has to be processed by the Scheduler, it is possible to specify a special-purpose initialization event. Initialization events are queued either by Runners or Composites. Alternatively, they may be sent directly by the Scheduler at idle time (as explained in Section 5.14). It is possible to configure the architecture to never send initialization events, to send them during the pre-process phase, or to send them during the process phase of Runners and Composites. Initialization events may be avoided if the Scheduler is already explicitly started with other types of events, which may be fired, for instance, from the outside of a Composite, from the pre-process phase of Units or from the GUI.

Listing 5.5 provides a simplified version of the scheduling loop, which is mostly responsible for

- Regularly executing the application’s critical section (see Section 3.22) and verifying the presence of stop requests. If processing needs to stop, the scheduling loop is interrupted.

- Performing idle time operations, such as scheduler contributions (see Section 6.15 for more details) if there are no scheduled Units (the queue is empty).

- According to scheduling requests, resuming the execution of the Scheduler container, which is either a Composite or the Runner, or executing the process function of the next scheduled component, which is either a Unit or a Composite.
5.8. Schedulers

Listing 5.5: Simplified version of the Scheduler loop.

```c
for (;;) {
    // Forces the critical section and exits
    // on application stop request
    if (forceCriticalSection(sched->application))
        break;

    // Performs idle time
    while (isIdle(sched)) {
        if (forceCriticalSection(sched->application)) {
            // Exits on application stop request
            break;

            // Idle time operations (scheduler contribution)
            performIdleTime(sched->application);
        }
    }

    // Exits on application stop request
    if (isStopped(sched->application))
        break;

    // Resumes the container or executes the component
    // Components may miss the process function
    currentComponent = getNextScheduledComponent(sched);
    if (currentComponent == sched->container) {
        sched->resumeContainer = 0;
        break;
    }

    if (currentComponent->process != NULL)
        currentComponent->process(currentComponent);
}
```

Depending on the information provided at the DSPE model level, variants of the Scheduler loop may be generated. For example, idle-time management may vary depending on the type of components contained in the Scheduler. If Composites with work to perform are present, the idle-time loop may be interrupted. More information about idle time management and inclusion of Composites in Schedulers is provided in Section 5.14.

Apart from the scheduling loop, which is the core part of Schedulers, the source code of Schedulers also includes most of the Configurations’ functionalities for initialization and disposal of contained components. Additionally, all the support for connecting Event Gates by means of sendEvent() functions is provided.
5.9 State Unit Behaviours

State Unit Behaviours support the development of finite-state machines, which may be required by Software Units inside Schedulers. State Unit Behaviours may be either State Implementations or State Block Optimizations (see Section 3.16 for more information). A state machine reacts to incoming events by modifying its internal state and, depending on the performed state transition, executing some type of processing and firing output events. Therefore, State Unit Behaviours feature a special purpose internal state and a list of values it may assume.

State machines may be implemented directly in the process function of a State Unit Behaviour. The source code, with the isEventAvailable() function, has to test if an event is available in the input event queue. Then, it retrieves the event ID using the getEventID() function. On the basis of the returned ID, the source code accesses the event with the getEvent() function, performs processing operations, modifies the internal state and fires output events. Transit queues and arm queues support suspending processing until all the required information is available.

Furthermore, State Unit Behaviours provide the following most prominent functionalities:

1. The advanced event support described in Section 5.7, which allows transiting and arming events.

2. Automatic features for event management. More details will be provided in the next chapter.

3. A limited version of the support for execution pressure management in parallel processing applications. See Section 6.14 and Section 6.16 for more information.

These additional functionalities are mainly useful when using State Unit Behaviours in combination with Coprocessor Unit Behaviours and GP Unit Behaviours for the development of parallel stream-processing applications.

5.10 Automatic Support for Events

The event-driven infrastructure generated by DSPE provides automatic functionalities for event management. These features allow using events transparently with respect to the algorithms contained in Unit Behaviours. Separation of algorithms from the software architecture is fostered by the component-based development technique used in DSPE (see Section 2.3). Concerning events, even if it is possible to access their infrastructure and fire events, by means of getEvent() and postEvent() functions, from the algorithms explicitly, it is not necessary to do so. With the automatic support, the developer is free to implement the algorithms without worrying about the flow of events. The algorithm can read and write the event data with the
5.11 Special-Purpose Lifecycle Phases for Events

<table>
<thead>
<tr>
<th>Feature</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Auto transit</td>
<td>Incoming events are automatically appended to the transit queues.</td>
</tr>
<tr>
<td>Auto execute</td>
<td>It waits that at least an event for each involved input connection is available in the respective transit queue. In other words, it synchronizes events. Then, it automatically starts processing.</td>
</tr>
<tr>
<td>Auto load</td>
<td>Prepares the memory locations accessible from the Unit Behaviour. It loads the internal data shortcuts of the Unit Behaviour with values and references to the data contained in the events in transit. Furthermore, auto load prepares the output events that have to be sent after processing has finished and loads their data shortcuts.</td>
</tr>
<tr>
<td>Auto send</td>
<td>Output events are automatically fired by the component.</td>
</tr>
</tbody>
</table>

help of special-purpose data shortcuts (see Section 5.6 for details). The source code generated from the information available in the DSPE model automatically manages all involved events and related data shortcuts. The infrastructure has automatic features for transiting, preparing, and firing events, which free the user from managing events directly.

For automatically managing events, DSPE provides the features listed in Table 5.2.

When developing stream-processing applications with DSPE, these automatic features are mainly used in combination with the automatic features for lightweight tasks provided by the soft coprocessor infrastructure (see Section 6.10 for more details). However, they may also be used independently in situations where events are utilized with a different purpose than supporting parallel processing.

To allow flexible use of these automatic functionalities, the DSPE model supports their progressive activation or deactivation. The automatic features depend on one another in the following way: auto-load requires auto-execute, and auto-execute requires auto-transit. Auto-send is, instead, completely independent from the others. Furthermore, in the DSPE model, it is possible to exclude any Event Gate from being automatically managed. Section 8.16 provides more information about these DSPE model features.

5.11 Special-Purpose Lifecycle Phases for Events

As illustrated in Figure 5.7, State Unit Behaviours feature additional lifecycle phases (See Section 3.15), to allow the development of software components that perform processing even if the auto-execute condition is not met. The additionally introduced phases are described in Table 5.3.
Table 5.3: Additional lifecycle phases for events.

Pre-compulsory process (represented with a small rectangle in Figure 5.7) a phase present only in State Implementations that is executed, even if the test of the auto-execute condition returns false, before the compulsory process phase. This phase is similar to the initialize parameters phase described in Section 3.19 but associated with the compulsory process instead of the process phase. It mostly allows performing calculation on the incoming parameter values when block processing is enabled.

Compulsory process a phase present in State Implementations and State Block Optimizations. It is executed even if the auto-execute condition is not satisfied. This phase allows processing the data and providing results during all invocations of the State Unit Behaviour.

Post-compulsory Process (represented with a small rectangle in Figure 5.7) also present only in State Implementations. It is similar to the finalize parameters lifecycle phase described in Section 3.19. The post-compulsory process phase is executed after the compulsory process phase and is used mostly for calculating the resulting parameter values.

Figure 5.7: Lifecycle phases of State Unit Behaviours.
5.12 Events and Composites

Event Composites are special-purpose Composites (see Section 3.13) that allow transferring events between the Configuration/Scheduler containing them and the Configuration/Scheduler, which is instead contained. Like Event Units, Event Composites have Event Gates. As soon as a Composite has one of them, the Composite is automatically converted into an Event Composite. As with normal Gates, Event Gates are used for connecting the Composite both from the inside and from the outside.

However, to support block processing with different block sizes between the inside and outside of the Composite, the generated source features special-purpose event infrastructure. As
introduced in Section 3.21, block processing with different block sizes requires gate size propagation during application initialization and pointer artifices for managing the Gate memory area when processing. These solutions are used for Event Gates too. Furthermore, for Event Composites, event multipliers and event filters are additionally exploited:

**An event multiplier** clones each event at the input of the Event Composite. The number of clones that equals the ratio between output block size and input block size minus one is produced. According to the real-time policies described in Section 3.10, for Parameter Gates the contained information is copied. Clones for Data Gates contain, instead, a reference to a sub-portion of the data block (see Figure 5.8). Each clone also has a reference to the original event. This reference is used for preventing the original event from being recycled in the event pool until all clones have also been recycled.

**An event filter** suspends events at the output of an Event Composite as long as a sufficient number of them is available. The number of clones that equals the ratio between output block size and input block size is suspended. Then, a single output event is left free to go through. According to the real-time policies of Section 3.10, for Parameter Gates, the last received event is the one sent at the output. This solution allows forwarding the most recent instantaneous value of the parameter. Concerning Data Gates, the first event, which contains a reference to the beginning of the complete block, is instead left free to continue, as shown in Figure 5.8.

To support using different block sizes, special purpose infrastructure is also required in Event Units. The block of data is constructed with the largest block size, as it is done for normal Gates (see Section 3.19). Then, event clones with reference to sub-parts of the block are constructed for satisfying the block-size needs of the internal Configurations and Schedulers (see Figure 5.9). Inside Implementations and Block Optimizations, shortcuts are used to reference the memory regions that need to be written in each process phase. Shortcuts hide the complex infrastructure required for managing events and clones, which is mostly managed by `sendEvent()` functions installed by Composites on Units.

### 5.13 Events and Runners

Use of events is also supported in combination with Runners (see Section 3.14). Event Runners are special-purpose Runners (see Section 3.13) with support for events. As soon as a Runner has an Event Gate, it is automatically converted into an Event Runner. The generated infrastructure allows sending and receiving events as part of the configuration parameters exchanged with the critical section. Furthermore, the generated user interfaces provide support for firing events and for showing the information contained in the received events.
5.14 Mixing Configurations and Schedulers

In DSPE, by profiting from Composites, it is possible to use combinations of Configurations, Schedulers, and Coprocessor Schedulers in the same application. These possibilities are of help, for example, when designing and implementing applications, which combine different control and processing behaviours. While Configurations (see Section 3.12) are ideal for sequential processing, Schedulers (see Section 5.8) are more flexible when adapting the application execution to dynamically varying stream rates. Coprocessor Schedulers (see Section 6.7) facilitate, instead, efficient parallel stream processing.

To allow flexibly combining Configurations, Schedulers and Coprocessor Schedulers, special-purpose support is provided. In Schedulers and Coprocessor Schedulers, it is possible to use different policies for managing idle time and suspensions. Figure 5.10 provides an example with some of the possible situations. Depending on application requirements, it is possible to specify the following behaviours:

**Resume the execution of the scheduler container,** which may be a Configuration, another Scheduler or Coprocessor Scheduler. This type of behaviour is typical of schedulers encapsulated in Composites.

**Force the execution of contained Units or Composites,** which may be implemented with
Chapter 5. Event-Driven Infrastructure

Schedulers having waiting events in queues. This type of behaviour is mostly used if the scheduler is directly contained in the application Runner and contains Composites with further schedulers.

Perform the scheduler contribution. This is useful in combination with the soft coprocessor infrastructure when all schedulers contained in an application become idle. More details about the scheduler contribution are provided in Section 3.12.

Freeze the scheduler temporarily. This behaviour is mostly used if the scheduler has to wait for external information before continuing processing. For instance, it may be used when waiting for soft coprocessors (see Section 3.12 for more details).

Fire an error by using the error manager (see Section 3.23 and Section 3.24). This is useful if the scheduler is never supposed to go idle.

Reinitialize the scheduler by sending the special-purpose initialization event (see Section 5.8). This type of behaviour is useful if the scheduler may eventually terminate processing and the only way of interrupting idle time is by reinitializing the scheduler.

Busy-waits by spinning in a short loop and continue testing if events are present. In other words, no operation—different from what is normally done—is performed. This approach is useful when maximizing performances, in particular in combination with soft coprocessors (see Section 3.12 for more details).

Two types of idle time behaviour are allowed. Local idle time tests only if the scheduler is idle. No guaranty is provided that contained Composites also have empty event queues. Instead, global idle time is performed by additionally verifying that all contained Composites are also idle. Apart from the two types of idle time, the described special-purpose behaviours may also be used when the scheduler resumes after a freeze (see Section 3.12 for more information about freeze). Three situations of wake-up after freeze are possible:

1. **Wake-up with a timer.** This type of solution may be used in applications that need to execute at fixed time intervals.

2. **Wake-up when the user interacts with the user interface.** Very useful in interactive applications.

3. **Wake-up when a special-purpose task-ready event has been received** (see Section 6.4).
As introduced in Section 4.9, to obtain significant performance improvements from multi-processors, it is important that the application execution is load balanced and occupies the available processing resources as fully as possible. For this reason, DSPE features a dynamic-load balancing infrastructure for parallel stream processing, which is possible to configure and generate specifically for each application. This infrastructure is composed of the following three main parts:

1. The *event-driven infrastructure* specialized for parallel stream processing, presented in the previous chapter.

2. A *soft coprocessor infrastructure*, which supports execution of lightweight task by means of work sharing and work stealing.

3. An *accelerator infrastructure* that supports hardware platforms featuring heterogeneous types of multiprocessors. These are, for instance, combinations of multi-core CPUs and many-core GPUs.

This chapter discusses the DSPE soft coprocessor infrastructure in detail. Information is provided about how this infrastructure integrates and enhances the DSPE event-driven infrastructure presented in the previous chapter. The soft coprocessor solution adopted in DSPE mainly supports parallel processing on general-purpose multi-core processors. Special-purpose many-core processors are, instead, supported by the accelerator infrastructure, which stands on the shoulders of the soft coprocessor infrastructure as discussed in the next chapter.

Essentially, the soft coprocessor infrastructure of DSPE provides functionality for dynamically distributing application-processing partitions to the available processor cores. The generated infrastructure takes advantage of events and features a parallel processing approach, which is specialized for real-time stream processing. Lightweight tasks are used as a means for parallel execution. Lightweight tasks are portions of the work the application has to perform, which can be executed independently. In DSPE, soft coprocessors are the blind executors that process lightweight tasks concurrently.
Chapter 6. Soft Coprocessor Infrastructure

6.1 Abstract Types of the Soft Coprocessor Infrastructure

In a way similar to the abstract types provided for the base infrastructure (see Section 3.6) and for the event-driven infrastructure (see Section 5.2), there are also structs for abstract types of the soft coprocessor infrastructure. These structs are the root of the hierarchy of objects, which are extended by specific DSPE elements in the generated source code.

Most of the abstract types of the soft coprocessor infrastructure extend the abstract data types of the event-driven infrastructure. The most important DSPE elements are contained in Table 6.1

| Table 6.1: DSPE elements of the soft coprocessor infrastructure. |
|------------------|---------------------------------------------------------------|
| Coprocessor Tasks | (abstract type DSPETask) are structs, which associate an algorithm to be executed with the data the algorithms have to process and the memory regions that have to be used for storing results. |
| Coprocessors     | (abstract type DSPECoprocessor) are independent executors of Coprocessor Tasks. Each Coprocessor has its own processing resources like an independent thread of execution. |
| Coprocessor Units | (abstract type DSPECoprocessorUnit) are special-purpose versions of Queue Units (see Section 5.5) that feature functionality for task management. Each Coprocessor Unit has access to a coprocessor queue of the soft coprocessor infrastructure. Coprocessor queues are buffers shared between software components and the soft coprocessor infrastructure, which are used for dispatching ready-to-process tasks to coprocessors. |
| Coprocessor Unit Behaviours | (abstract type DSPECoprocessorUnitBehaviour) are Unit Behaviours with programmatic access to functionality for preparing, queuing, and consuming tasks. There are two types of Coprocessor Unit Behaviours: Coprocessor Implementations and Coprocessor Block Optimizations. |
| Coprocessor Schedulers | (abstract type DSPECoprocessorScheduler) are dedicated versions of Schedulers, which provide integration with the soft coprocessor infrastructure. |
| Task-ready Events | (abstract type DSPECoprocessorEvent) are special-purpose Events, used to inform Coprocessor Units that tasks have been executed. |
6.2 Lightweight tasks, Coprocessor Units, and Soft Coprocessors

In DSPE, tasks are structs that extend the abstract data type Coprocessor Task. Each task is made of

- A function pointer of an algorithm to execute.
- Values required for processing that are copied from parameters or from the Unit’s internal states.
- References to memory regions of data to process and references to memory regions where results have to be written.

The struct of the abstract type DSPECoprocessorTask and an example of the child struct of a task are provided in Listing 6.1.

Listing 6.1: Struct of the DSPECoprocessorTask abstract type and example of the child struct of a task.

```c
struct DSPECoprocessorTask {
    DSPEElement element;
    void (*execute)(DSPECoprocessorTask *task);
};

struct EnergyTask {
    DSPECoprocessorTask parent;
    int blockSize;
    EnergyTasks *next;
    unsigned int processed;
    float energyTot;
    CubeImage_Event *dataIn;
    CubeImage_Event *dataOut;
    int *nPhase;
    StiffnessMatrix *delta;
};
```
Chapter 6. Soft Coprocessor Infrastructure

In DSPE, tasks are produced by Coprocessor Units, which are special-purpose versions of Software Units, or by the contained Coprocessor Unit Behaviours. Typically, each task contains a portion of the stream to be processed. Therefore, the Coprocessor Units generated with DSPE continuously produce tasks while the stream is flowing through the application and while it is being processed (see Figure 6.1).

![Figure 6.1: Production of lightweight tasks performed by Coprocessor Units.](image)

Normally, the functions executed by soft coprocessors when processing tasks are the process functions of Coprocessor Unit Behaviours. The function pointers are automatically installed when tasks are initialized. Therefore, in DSPE, tasks are snapshots of a predefined size of the processing activities performed by Coprocessor Units. These snapshots, which have their own independent state are executed by soft coprocessors, instead of by the main thread of the application.

Coprocessors are task executors. Each Coprocessor owns an independent thread and processes the assigned tasks blindly. It executes the function referenced by the task on the stored values or on values stored in the contained data references. Coprocessors have no notion about the executed process function and about the processed information.

### 6.3 Design Patterns Used in the Soft Coprocessor Infrastructure

The following main design patterns are used in the soft coprocessor infrastructure:

**Thread pool** (also **task pool** or **work pool**) [52, 134]: the soft coprocessor infrastructure of DSPE is an enhanced thread pool, which allows managing either a predefined or a dynamic number of threads. Thread pools are advantageous, because they allow saving unnecessary thread creation and destruction costs. Furthermore, with thread pools, it is possible to avoid over-threading (more threads than available cores), which may cause frequent context switches and consequent performance degradation. Thread pools are normally used for processing lightweight tasks (small units of work). Therefore, both of the terms “task pool” and “work pool” are alternatively used for this design pattern. If many sufficiently small tasks are processed, thread pools facilitate load balancing and better scalability.

**Master-worker** (also **producer-consumer**) [96, 108, 18]: the name “soft coprocessor” has been used in DSPE because the software architecture it generates implements the master-worker
6.4. Integration of Soft Coprocessors with the Event-Driven Infrastructure

pattern. In this pattern, there are two actors: masters, which produce and queue work to perform, and workers, which consume the produced work. In DSPE, Coprocessor Units inside Coprocessor Schedulers play the role of the masters, whereas Coprocessors are slave consumers.

Shared queue [96, 108]: Coprocessor Units and Coprocessor Schedulers communicate with soft coprocessors by means of shared queues. These queues have the role of synchronizing the parts of the application that execute concurrently. Several types of shared queues may be generated by DSPE. For instance, non-blocking queues based on atomic operations (see Section 4.10) may be used.

Event-based coordination [96]: Because of the integration of the soft coprocessor infrastructure with the event-driven infrastructure of DSPE, the execution of the lightweight tasks produced by the software components is performed with event-driven scheduling. As a consequence, the parallel processing software components of the application are coordinated in an event-based way, which depends on how the data streams flow between the software components.

6.4 Integration of Soft Coprocessors with the Event-Driven Infrastructure

The soft coprocessor infrastructure and the event-driven infrastructure integrate as follows:

Coprocessor Schedulers and the contained Coprocessor Units play the role of the master and perform scheduling and task queuing operations. For this purpose, Coprocessor Unit Behaviours, which are included in Coprocessor Units—as a complement to the support for transiting-dismissing input events and for arming-posting output events (see Section 5.7)—feature special-purpose support for creating, queuing, and dismissing lightweight tasks (see Section 6.8 for more details).

This special-purpose support has access to the task pool (see the next section for more details), which allows creating new tasks or recycling unused tasks. When the task pool makes an empty task available, Coprocessor Units and Coprocessor Unit Behaviours can fill the task by copying parameter and internal state values of the Coprocessor Unit Behaviour, or by assigning memory references to memory regions of transiting and armed events. Therefore, tasks also act as aggregators of events and are responsible for delivering these events to soft coprocessors. As soon as a software component has a ready task, it appends the task to a shared coprocessor queue. These queues are managed by the soft coprocessor infrastructure and are used for delivering the tasks to an idle coprocessor (see Figure 6.2).

When processing has finished, each coprocessor fires a special-purpose task-ready event to the Coprocessor Unit where the task originated. The task-ready event contains a reference to the processed task and signals that results are available. Subsequently, when consuming the task-ready event, if the task is the least recently queued, the task is dismissed and can
Figure 6.2: Integration of soft coprocessors with the event-driven infrastructure. Operations are performed as follows: 1. events are exchanged between Software Units; 2. when events are received by Coprocessor Units, tasks are prepared and queued; 3. tasks are processed by soft coprocessors; 4. task-ready events are sent to the Coprocessor Units.

be recycled in the task pool. At the same time, the Coprocessor Unit dismisses the transiting events and fires the armed events. A special-purpose task buffer is used to keep track of the order of the tasks. If a task-ready event is received for a task that is not the least recently queued, the release of the task is postponed.

The processing cycle, from arrival of input events and task instantiation, to dispatch of output events and task release, is completely asynchronous. There are no needs for extras, such as special-purpose threads, which wait until processing of tasks completes. After a task has been queued, the Coprocessor Scheduler and the contained Units are free to continue processing and, for instance, queue more tasks to the soft coprocessor infrastructure.

6.5 Memory Management: Task Pools

In a way similar to the event pools featured by the event-driven infrastructure (see Section 5.4), the soft coprocessor infrastructure features task pools for recycling tasks. Each time a new task is needed, it is requested by the Coprocessor Unit or by the contained Coprocessor Unit Behaviour to the task pool. If a recycled task is available, the pool releases it. If, instead, the pool is empty, a new task is constructed. After processing, when tasks are dismissed, they
6.6. Task Immutability and Restricted Access Time

In DSPE, tasks are seen as specialized types of events. Like all other events, tasks contain information, such as the data of the streams and parameters exchanged between processing elements. However, tasks may also contain references to events or references to the memory regions of events. Furthermore, unlike other types of events, tasks also contain a function pointer, which represents processing operations that have to be performed. Similar to all other events, tasks in DSPE are considered immutable (see Section 4.11 and Section 5.3 for more details). However, immutability of tasks has two phases:

The first phase starts after Coprocessor Units have finished preparing the tasks. This phase involves only input references. After this phase has started, all the memory regions, of which the references have been copied in the task and are used for processing, have to be considered read-only. This is particularly important, because more than one parallel executing task may contain the same references, which may, therefore, be accessed simultaneously by more than
The second phase starts after the task has been processed and all results have been written in the task. Mostly, this second phase involves the output references. After this phase has started, the memory regions of results, of which the references are copied in the task from armed events, have to be considered read-only. This phase should last, at least until the special-purpose event—signalling that processing of the task has finished—has been received by the Coprocessor Unit Behaviour.

As a consequence, in general, the time interval between when tasks are queued to coprocessors and results are back is considered a restricted access time. During this time span, neither the Coprocessor Unit that produced the task nor other external parts of code of the application are allowed to perform write operations on the memory regions referenced by the task. The task is the only one to have access to the output memory regions. On those memory regions, even read operations are forbidden from the outside. Read operations on the input memory regions, done by tasks or other parts of code, are permitted instead. Obviously, input memory regions remain read-only, including for tasks, even during the restricted task-access time.

It is important to respect the restricted access time, in particular if algorithms execute partly inside Unit Behaviours and partly inside tasks produced by the Unit Behaviours. In these situations, the same memory regions could be accessed concurrently. Restricted access time grants thread safety and reduces the synchronization overheads to the minimum.

### 6.7 Coprocessor Schedulers

Coprocessor Schedulers extend and specialize the behaviour of Schedulers (see Section 5.8). Like Schedulers, they track the flow of events and execute Units. In addition, Coprocessor Schedulers integrate with the soft coprocessor infrastructure.

Coprocessor Schedulers mainly feature

1. The queue of processed tasks that are sent back by the soft coprocessor infrastructure.
2. An enhanced version of the scheduling loop, which takes care of the processed tasks.
3. The infrastructure for sending special-purpose, task-ready events to Coprocessor Units.

Listing 6.2 provides a simplified version of the scheduling loop for the Coprocessor Scheduler. Besides the functionalities described in Section 5.8, the scheduling loop is responsible for emptying the queue of processed tasks and sending corresponding task-ready events to the Coprocessor Units that originated the tasks.
Listing 6.2: Simplified version of the scheduling loop of Coprocessor Schedulers.

```c
for (;;) {
    // Forces the critical section and exits
    // on application stop request
    if (forceCriticalSection(sched->application))
        break;

    // Sends events for ready tasks
    currentTask = getCurrentTask(sched->outQueue);
    while (currentTask != NULL) {
        sendTaskReadyEvent(currentTask);
        currentTask = getCurrentTask(sched->outQueue);
    }

    // Performs idle time
    while (isIdle(sched)) {
        if (forceCriticalSection(sched->application)) {
            // Exits on application stop request
            break;

            // Idle time operations (scheduler contribution)
            performIdleTime(sched->application);

            // Sends events for ready tasks
            currentTask = getCurrentTask(sched->outQueue);
            while (currentTask != NULL) {
                sendTaskReadyEvent(currentTask);
                currentTask = getCurrentTask(sched->outQueue);
            }
        }
    }

    // Exits on application stop request
    if (isStopped(sched->application))
        break;

    // Resumes the container or executes the component
    // Components may miss the process function
    currentComponent = getNextScheduledComponent(sched);
    if (currentComponent == sched->container) {
        sched->resumeContainer = 0;
        break;
    }
    if (currentComponent->process != NULL)
        currentComponent->process(currentComponent);
}
```
By means of Coprocessor Units, Coprocessor Schedulers may delegate the execution of algorithms to soft coprocessors. However, they are not obliged to do so for every contained software component. For algorithms that execute fast, there may be no benefit, when also considering the overheads introduced by the soft coprocessor infrastructure. Therefore, as with Schedulers, in some situations the execution of the processing algorithms partially remains the direct responsibility of the Coprocessor Schedulers. Section 6.13 provide more details about this topic, in particular concerning the impact on the resulting performances.

6.8 Coprocessor Unit Behaviours

There are two types of Coprocessor Unit Behaviours: Coprocessor Implementations and Coprocessor Block Optimizations. These Unit Behaviours extend and specialize the respective State Unit Behaviours by providing additional processing phases and support for preparing and queuing lightweight tasks to coprocessors. Listing 6.3 provides the interfaces of the available functions for preparing, scheduling and disposing lightweight tasks.

![Figure 6.3: Lifecycle phases of Coprocessor Unit Behaviours.](image)
In Figure 6.3, the main lifecycle phases that are present in Coprocessor Implementations and Coprocessor Block Optimizations are illustrated with large rectangles, whereas small rectangles are used for the additional phases present only in Coprocessor Implementations. As introduced in Section 3.19, these additional phases are present in Implementations mainly to deal with parameters when performing block processing. In Coprocessor Unit Behaviours, start-up, pre-process, compulsory process, post-process, and shutdown phases are identical to the phases provided by State Unit Behaviours (see Section 5.11). Instead, the process phase, together with its initialize parameters and finalize parameters phases (represented, respectively, with “Init” and “Fin” in Figure 6.3), are executed by soft coprocessors as an alternative to direct execution by the Coprocessor Unit Behaviour. As a consequence, Coprocessor Unit Behaviours feature two additional lifecycle phases (see Table 6.2).

The additional phases of Table 6.2 are mostly responsible for delegating processing work to soft coprocessors. This delegation introduces asynchrony in the execution of the Coprocessor Unit Behaviours. The dismiss-task phases are executed only when the Coprocessor Unit Behaviour receives special-purpose task-ready events. Consequently, it may happen that the arm-task phase and its preceding phases execute many times before a dismiss-task phase and its subsequent phases are performed.

Listing 6.3: Functions for managing tasks provided by Coprocessor Unit Behaviours.

```c
DSPETask* getTaskFromPool(DSPEUnitBehaviour *context);
DSPETask* createTask(DSPEUnitBehaviour *context);
void initTask(DSPEUnitBehaviour *context, DSPETask *task);
void transferGates(DSPEUnitBehaviour *context, DSPETask *task);
void queueTask(DSPEUnitBehaviour *context, DSPETask *task);
DSPETask* getProcessedTask(DSPEUnitBehaviour *context);
void releaseTask(DSPEUnitBehaviour *context, DSPETask *task);
void destroyTask(DSPEUnitBehaviour *context, DSPETask *task);
```

<table>
<thead>
<tr>
<th>Arm-task</th>
<th>a lifecycle phase mostly used for preparing and queuing tasks.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dismiss-task</td>
<td>a lifecycle phase used for consuming the task and sending the associated events.</td>
</tr>
</tbody>
</table>

Table 6.2: Additional lifecycle phases of Coprocessor Unit Behaviours.
Figure 6.4: Processing of tasks of shared and exclusive queues. Tasks in shared queues may be processed concurrently.

6.9 State-Less vs. State-Full Algorithm Support

As introduced in Section 4.5, Software Units may encapsulate state-full algorithms that require sequential processing. For this purpose, DSPE is able to generate special-purpose versions of the soft coprocessor infrastructure. This is done by taking advantage of a feature available in the DSPE model for all internal states of Unit Behaviours. In the DSPE model, it is possible to differentiate among

**Read-only states:** this kind of state is immutable. It is never written after initialization.

**Temporary states:** this kind of state is used only during one process round, which corresponds to a restricted access time interval (see Section 6.6). At the beginning of each process round, the state is reinitialized.

**Permanent states:** this kind of state is used for storing information required during more than a processing round. Normally, these states are modified continuously during processing and contain the memories of software components, such as delay lines in digital filters.

Software components are considered state-full if they contain permanent states. As a consequence, the DSPE generators use the mentioned model information to produce infrastructures with different behaviours (see Figure 6.4 for an example). If a Coprocessor Unit Behaviour contains only read-only or temporary states, parallel execution of the tasks, which are queued by the Coprocessor Unit, is allowed. The task queue is shared by all coprocessors, which may concurrently retrieve and process the contained tasks. Even if a coprocessor is busy performing a task from the task queue, another coprocessor is allowed to start the subsequent
task. Instead, if there are permanent states, the task queue of the Unit Behaviour is considered exclusive. Coprocessors are allowed to start processing a task from an exclusive queue, only if other coprocessors are not executing preceding tasks from the same queue. Parallel execution of the tasks is prevented.

For state-less components, it is also possible to generate an alternative version that features more than one shared queue. The Coprocessor Units, instead of always appending the tasks to the same queue, append them to one of the available queues in round-robin manner. All these queues are visible from the soft coprocessors, which are allowed to consume them in parallel. This version helps in situations where a single shared queue becomes a bottleneck and, in addition, facilitates work stealing (see Section 6.12 for more details).

Coprocessor Unit Behaviours and tasks access the different types of states generated in the source code as follows:

• **During start-up and pre-process phases**, Coprocessor Unit Behaviours may access read-only and permanent states with write permission. This approach allows default initialization of states (A and B in Figure 6.5). During these phases, Coprocessor Unit Behaviours have their own temporary states (F in Figure 6.5).

• **During arm-task and dismiss-task phases**, each individual temporary state associated with a task may be modified by the Coprocessor Unit Behaviour to perform value initialization (G in Figure 6.5). Additionally, the Coprocessor Unit Behaviour may use its own temporary states (F in Figure 6.5) or access read-only states with read permission (A in Figure 6.5). During these phases, all permanent states are not visible from the Coprocessor Unit Behaviours because soft coprocessors may be contemporaneously executing a task that performs write operations.

Figure 6.5: Internal states of tasks and Coprocessor Unit Behaviours.
• During processing, read-only states may be accessed by all tasks, but only with read permission (C in Figure 6.5). Permanent states are accessible only by one task at a time (D in Figure 6.5), whereas each individual temporary state may be accessed only by the task in which it is contained (E in 6.5).

The data shortcuts generated as part of the source code of Unit Behaviours (see Section 3.16) facilitate the correct use of the states from the different processing phases. For example, shortcuts to the permanent states are not available for the arm-task and dismiss-task phases.

6.10 Automatic Support for Tasks

As introduced in Section 5.10, the DSPE generators are able to produce automatic management features for events that relieve the user from administering events explicitly. Similarly, there are automatic management features for tasks. Even if, in the functions of the Coprocessor Unit Behaviours, it is possible to create and queue tasks explicitly, it is also possible to leave this responsibility to the DSPE-generated infrastructure. Automatic features produced by the DSPE generators free the user from managing tasks directly. As a consequence, the soft coprocessor infrastructure remains transparent with respect to the algorithms.

To manage lightweight tasks automatically, DSPE provides the additional features of Table 6.3. These automatic functionalities require activation of the automatic support for events described in Section 5.10. When all automatic features are enabled (see Figure 6.6), algorithms inside Coprocessor Unit Behaviours are completely transparent, both from events and from tasks.

Figure 6.6: Support for automatically managing events and lightweight tasks.
6.11. Coprocessor Manager

Figure 6.7: Dependencies of the automatic features for events and tasks.

Table 6.3: Automatic features for managing tasks.

<table>
<thead>
<tr>
<th>Feature</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Auto transfer</td>
<td>requests a task from the task pool and copies all the values and references from the internal shortcuts to the inputs and outputs of the task. After auto transfer completes, the task is ready to be queued.</td>
</tr>
<tr>
<td>Auto task management</td>
<td>after a lightweight task has been prepared (with auto load and auto transfer), it automatically dispatches it to soft coprocessors. Later, after the task has been processed, it recycles the task in the task pool.</td>
</tr>
</tbody>
</table>

Figure 6.7 provides the overview of the dependencies existing between all available automatic features. As for the automatic support for events, Gate omissions may be specified to bypass some Event Gates from being managed with the automatic support for tasks.

Mostly, these automatic features facilitate the development of conventional stream-processing components that process the incoming data streams on a regular basis. Before execution, the component waits on an event for each input Gate (except omitted Gates) by utilizing transit queues. Then, the component queues a task, which is executed by the soft coprocessor. Normally, the same algorithm is executed all along processing. Afterward, the component fires an output event for each of its output Gates (except omitted Gates). On the other hand, to allow flexible development of out-of-the-ordinary software components—which, for instance, perform conditional processing or conditional delivery of output events—DSPE allows enabling or disabling of the individual automatic functionalities to intervene with the management of events and tasks programmatically.

6.11 Coprocessor Manager

A significant part of the source code of the DSPE parallel stream-processing run-time, including the soft coprocessor infrastructure and the accelerator infrastructure (see Chapter 7), has been isolated in a manager called coprocessor manager. As introduced in Section 3.23, managers facilitate code reuse, evolution, and portability.

The coprocessor manager features the source code for

1. All input and output queues to coprocessors.
Chapter 6. Soft Coprocessor Infrastructure

2. The infrastructure for efficiently managing these queues.


Listing 6.4 and Listing 6.5 provide simplified versions of the source code used for executing tasks in coprocessors. In the sample code, the main lock operations are provided to inform about the atomic execution requirements. However, with the DSPE generators, it is possible to generate other versions of the code in Listing 6.4 and Listing 6.5. These versions replace locks with atomic operations or interleave lock operations and other operations differently. In particular, these alternative versions profit from a more tolerant approach, which allows relaxing locking at the expense of some aimless access to the queues that may happen in the event of unlucky synchronisation. Depending on the nature of the generated application, these versions may be more efficient than the rigorous lock-based version.

Listing 6.4: Main execution loop of soft coprocessors.

```c
void coprocessorRoutine(void* args) {
    DSPECoprocessor *coprocessor = (DSPECoprocessor*) args;
    DSPECoprocessorQueue *inQueue = NULL;

    // Executes continuously until suspended or terminated
    for (;;) {
        // Retrieves a task queue from the pool
        // and calls the processTask() function
        lockQueuePool();
        if (queuePool->numQueues == 0) {
            unlockQueuePool();
        } else {
            inQueue = queuePool->getQueue();
            unlockQueuePool();
            processTask(coprocessor, inQueue);
        }

        // Manages thread suspension and termination
        lockThreadPool();
        if (threadPool->state == ACTIVE) {
            unlockThreadPool();
        } else if (threadPool->state == SUSPENDING) {
            addThreadToPool(coprocessor);
            suspendThread(coprocessor);
            unlockThreadPool();
        } else if (threadPool->state == TERMINATING) {
            unlockThreadPool();
            break;
        }
    }
}
```
6.11. Coprocessor Manager

The source code of Listing 6.4 and Listing 6.5 is simplified and, consequently, many operations performed in the real code have been removed for comprehensibility. Nonetheless, the code of the coprocessor routine execute fast and avoid introducing unnecessary overheads when parallel stream processing.

Listing 6.4 features the code part that manages input queues and coprocessors. Instead, the source code of Listing 6.5 executes the task, releases the input queue, and appends the task to the output queue. For shared input queues, immediate release is performed. Exclusive queues are instead released after the task has finished processing.

Listing 6.5: Source code part responsible for processing tasks in soft coprocessors.

```c
void processTask(DSPECoprocessorQueue *inQueue) {
    DSPECoprocessorQueue *outQueue = NULL;
    DSPECoprocessorTask *task = NULL;

    // Retrieves a task and releases the queue if shared
    lockQueue(inQueue);
    task = getTaskFromQueue(inQueue);
    if (!inQueue->exclusive && inQueue->numTasks > 0)
        queuePool->releaseQueue(inQueue);
    unlockQueue(inQueue);

    // Processes the task
    task->execute(task);

    // Release the queue if it is exclusive
    lockQueue(inQueue);
    if (!inQueue->exclusive && inQueue->numTasks > 0)
        queuePool->releaseQueue(inQueue);
    unlockQueue(inQueue);

    // Sends the processed task to the scheduler
    coprocessorUnit = task->container;
    scheduler = coprocessorUnit->container;
    outQueue = scheduler->getCoprocessorQueue(scheduler);
    queue->addTaskToQueue(outQueue, task);
}
```

As part of the different variants that may be generated for Listing 6.4 and Listing 6.5, it is possible to activate support for simultaneously processing all tasks present in the input queue instead of releasing the queue at each processing round. Similarly, it is possible to enable support for simultaneously consuming a number of tasks proportional to the number of tasks contained in the queue or proportional to the size of the queue. These variants allow adjusting the ways tasks are consumed for fitting application-specific needs.
Chapter 6. Soft Coprocessor Infrastructure

6.12 Work Sharing and Work Stealing

To allow exploiting different types of task scheduling, in DSPE it is possible to utilize two forms of load balancing, which are based on the approaches of work sharing and work stealing. The generated infrastructures for these approaches differ in many details.

With work sharing, all input queues that have at least a task to process and are not already operated by a coprocessor are stored in a centralized work-share pool accessible by all coprocessors (see Figure 6.8). This pool is implemented with a queue of input task queues. Therefore, in the work-sharing version, coprocessors consume tasks by first removing the task queue at the head of the work-share pool and then removing one or more tasks from the queue. DSPE uses the approach of always removing the first queue from the pool because this method was shown to be the most effective. This approach consumes a short execution time that remains constant and predictable even if the work-share pool contains many queues. Other approaches may instead require walking along the queue with consequent overheads, which may be unsustainable when parallel stream processing.

Consequently, the solutions provided by DSPE for preferential consumption of the queues focus on the queue insertion policy. Indeed, after the coprocessors have completed all the required operations with a task queue, the queue is re-added to the work-share pool, depending on the absolute or relative number of contained tasks and depending on the queue priority. The following approaches are available:

**Always append:** queues are always re-added at the end of the pool queue. This approach is free of additional overheads and its infrastructure consumes a constant execution time. Furthermore, this approach limits the alterations that may appear in the response jitter of the application as a consequence of data buffering (see Section 6.14). However, in most situations it provides weak performances because there is no possibility of preferential consumption.

**Exact position:** the exact position of the task queue in the pool queue is found, by comparing the number of its contained tasks with the number of tasks of other task queues in the pool and by considering the queue priorities. This approach has the greatest potential in terms of performances, because it is possible to preferentially process queues, and, as a consequence, minimize the amount of queue overloads. However, the overhead required for traversing part or all the pool may be significant and unpredictable when the work-share pool contains many queues.

**Prepend if greater:** is a compromise between the “always append” and the “exact position” approaches. The task queue is appended at the end of the pool queue, unless its priority and number of contained tasks are not greater than the number of tasks contained in the queue at the head of the pool. If so, the queue is alternatively added at the beginning of the pool queue. The advantage of this approach over the exact position approach is a constant and predictable overhead. As a consequence, in most situations this approach provides the best performance.
In DSPE, the queue priority is optional information, which is derived in several ways, either from the information provided in the model or from an execution profile of the application. For instance, it is possible to assign a higher priority to software components that acquire or synthesize the data streams. This approach favours higher application execution pressure (see Section 6.14 for more details).

Compared with alternative approaches utilizable for scheduling tasks (for an example see the work-stealing solution described below), the work-sharing approach featured by DSPE uses a simple and lightweight infrastructure, introduces negligible overheads and, therefore, is well suited for real-time stream processing. However, in some situations work sharing may become a limit from the point of view of application scalability. If a very large number of parallel processing coprocessors concurrently access the centralized pool frequently, the pool may become a bottleneck.

Work stealing is an alternative approach, which solves the bottleneck problem of the work-share pool by decentralizing the pool. Task queues, instead of being collected in a unique location accessed by all soft coprocessors, are directly located with the coprocessors. Each task queue is assigned, per default, to one of them. Other coprocessors may steal task queues, or part of task queues, when they become idle.

The work-stealing approach provided by DSPE is specialized for stream processing. Therefore, care has been taken to use technical solutions, which introduce overheads that are low and possible to predict. However, the infrastructure used for work stealing is more complex compared with the infrastructure required by work sharing; consequently, there are some additional costs to account for. Nonetheless, depending on the nature of the developed application and used execution environment, work stealing may feature better scalability because of less thread contention. Furthermore, work stealing facilitates locality of processing.
One of the major challenges in work stealing is how to perform the steal. Many work-stealing approaches that are documented in the literature [118, 72, 119, 11, 68, 91] are based on data structures, such as trees, and the tasks are stolen randomly. Instead, in DSPE, the following approach has been adopted. All the available soft coprocessors are chained in a doubly linked list. This list is constructed at initialization time and is never modified. As a consequence, it is read-only and allows concurrent access without locking. Each soft coprocessor maintains its local pool of task queues. “Neighbour stealing” is performed: when a soft coprocessor becomes idle, it steals a complete queue from its neighbour soft coprocessors in the doubly linked lists (see Figure 6.9). To favour locality of processing, task lists are stolen by respecting simple rules. For instance, the last processed queue, either shared or exclusive, is stolen with low priority to favour locality of processing. To facilitate the stealing of only a part of a shared queue, it is possible to generate the version that uses more than one shared queue for each state-less component (see Section 6.9 for more information).

With the work-stealing approach adopted in DSPE, an important detail regards the initial assignment of task queues to soft coprocessors. During the pre-process lifecycle phase, it is possible to assign task queues to coprocessors randomly and to leave the responsibility of distributing the queues to the work-stealing mechanism at run time. However, this solution is not always efficient. By already distributing the task queues in a balanced way, it is possible to improve the behaviour of work stealing, at least during the first processing rounds of the application. Therefore, DSPE allows the generation of an alternative version that initially distributes the queues on the basis of their characteristics. Similarly, it is possible to generate a version that distributes the queues by using the information contained in execution profiles of the application. For instance, this version tries to distribute the most time-consuming queues as distantly as possible in the doubly linked list of soft coprocessors.

Figure 6.9: Work stealing is performed by stealing the tasks of neighbour soft coprocessors.
6.13 Long-Running and Blocking Tasks

In DSPE, the soft coprocessor infrastructure may be configured to satisfy different needs and to help obtain performance improvements. Mostly, lightweight tasks and soft coprocessors ease load balancing and improve the application’s scalability. Coprocessors execute operations blindly. As long as tasks are queued by components and coprocessors have work to do, coprocessors should be able to keep the available parallel processing resources busy. If a coprocessor executes a task lasting less than the type of task performed by another coprocessor, then, in subsequent execution rounds, the coprocessor will probably pick up a task that lasts longer, consequently balancing core occupation. See Figure 6.10 for an example of a possible interleaving.

![Diagram](image)

Figure 6.10: Example of a possible interleaving of lightweight tasks. Bars t1, t2, ..., t6 represent the execution times of lightweight tasks for an execution with 4 coprocessors.

However, there are situations, where performance improvements may not be easy to obtain. For instance, unbalanced situations may occur, if the task queues get empty, if the streams have an irregular or low incoming rate, or if coprocessors execute operations implying wait states, such as an access to a file. Short momentary load imbalances should not excessively compromise performances, which may instead be seriously obstructed by repetitive or long-lasting instabilities. Task queues may get empty if the application execution pressure is insufficient (see the next section for more details). This may be caused by data dependencies or bottleneck components as introduced in Section 4.5.

However, risks of persistent non-uniform executions may also occur, if coprocessors execute long-running or blocking tasks. Ideally, lightweight tasks should execute in 1 to 10 ms or even less, depending on the acceptable application latency. However, long-running or blocking tasks are sometimes needed by applications that combine control and processing behaviours. In DSPE, long or never-ending tasks are allowed but should be carefully managed because they prevent soft coprocessors from contributing to the overall load balancing. If most of the available soft coprocessors are occupied with these types of tasks, the performance improvements obtained from the soft coprocessor infrastructure may be compromised. Furthermore, care is required with the exclusive queues of state-full components because the queued tasks may remain blocked. As an alternative, long-running operations, blocking operations, or control operations related to time-critical requirements, can be managed with independent threads of execution, separately from the soft coprocessor infrastructure. Furthermore, to avoid slowing down the scheduler, operations using values and updating information on the user interface...
should also be managed independently by going through a critical section (see Section 3.23, Section 3.24 and Section 3.22 for more information about the thread manager and the critical section).

Ideally, to favour high streaming throughput and low processing latency, many small tasks of similar size are more adequate. Principally, appropriate task subdivision should be done at application design time, by using meaningful decomposition approaches (see Section 4.2) and by carefully developing the scheduler. As a rule of thumb, trying to keep the execution time of the scheduler as short as possible and delegating stream-processing operations to coprocessors result in good solutions. If possible, all control operations should be kept part of the scheduler, which, if required, can be executed with a dedicated high-priority thread (see Section 6.15 for more information). Tasks should be kept as small as possible. However, some limits are reasonable to decrease the execution overheads. Excessively small tasks may contrast the efficiency of the application. Therefore, light software components should be kept as part of the scheduler (as introduced in Section 6.7) or aggregated with other components. By decomposing the algorithms fairly and by using meaningful block sizes when splitting the streams (see Section 4.8 for details), it should be possible to maximize the performance improvements obtained with the soft coprocessor infrastructure of DSPE.

### 6.14 Application Execution Pressure

As introduced in the previous chapter and in Section 4.9, if task queues are emptied and coprocessors become idle, unbalanced execution and degradation of performance may result. Therefore, in parallel stream-processing applications, it is important that the execution pressure of the application remain as high as possible. High execution pressure means that the streams fluidly move through the software components, that task queues between component and soft coprocessors are sufficiently full, and that soft coprocessors do not get idle. To obtain a high pressure, design and implementation solutions are needed for scheduling tasks at regular rates and as constantly as possible.

Another problem that may appear when using the soft coprocessor infrastructure for parallel stream processing is represented by alterations in the response jitter of the application. Because of the data buffering performed in the task queues, the movements of the stream items between the software components may be similar to waves, in particular when using the “exact position” or the “prepend if greater” approaches (see Section 6.12). As a consequence, the results may be produced with an irregular rate at the output of the application. Resulting stream items may be aggregated in groups separated by production pauses. This type of side effect may compromise the real-time behaviour of the application. As a consequence, a high execution pressure has to be obtained by maintaining the application latency minimal and by limiting the alterations in the application response jitter as much as possible.

Mostly, a high execution pressure may be obtained with a fast and reactive scheduler that has some execution room as compared with the soft coprocessors. In this context, as introduced...
in the following sections, some technical solutions are of help. Furthermore, to reduce the impact on latency and response jitter, queue sizes have to be carefully tuned.

### 6.15 Scheduler Contribution and Scheduler as a Task

Tasks and soft coprocessors allow balancing the execution of the coprocessor threads on the available CPU cores. To even allow balanced execution of the coprocessor thread, the source code generated by DSPE provides two variants: scheduler contribution and scheduler as a task.

With the scheduler contribution, the scheduler is executed with a special-purpose thread, as in the default-generated source code version. However, if the scheduler thread executes faster than coprocessors and starts spinning idle, it performs the scheduler contribution and helps coprocessors by consuming tasks too. After a task has been consumed, it returns to its scheduling activities. To be effective, the scheduler contribution requires that the application contains at least one state-less Coprocessor Unit with sufficient queue capacity or more state-full Coprocessor Units than cores. Otherwise, the risk is high that the scheduler never has the opportunity to contribute.

Scheduler contributions may be of great help. However, they may introduce problems if tasks with considerable execution time are present. During the time period in which the thread of the scheduler performs the contribution, the scheduler stops delivering events and scheduling Units. As a consequence, task queues may become empty and the application execution pressure may reduce. In unlucky situations, all coprocessors may even finish the available work to do and sit idly waiting for the scheduler to resume.

Therefore, as an alternative, DSPE provides a solution that manages the scheduler as a task and executes it with the soft coprocessor infrastructure. With this solution, the scheduler is either shared by all coprocessors when performing work sharing or may be stolen like other tasks with work stealing. The infrastructure allows executing the scheduler with higher priority than other waiting tasks. Therefore, if the scheduler has a short execution time, this approach is advantageous. Chances that the scheduler executes frequently are maximized, and, as a consequence, the application’s execution pressure stabilizes. The disadvantages of the scheduler-as-a-task approach are represented by the additional overheads of the operations performed by soft coprocessors after task execution. These operations are required for verifying if the scheduler needs to execute and for launching its execution. Depending on the size of the tasks and the number of used threads, these overheads may not be negligible.

Scheduler contribution and the scheduler-as-a-task approach provide good compromises in terms of scheduler execution and occupation of the available cores. There are, however, exceptional situations, in which the best performances can be obtained with the default approach, by using only a dedicated and exclusive thread for the scheduler. Depending on the application needs and utilization context, analysis and experiments have to be conducted to
find the optimal solution.

6.16 Pull and Push Execution Pressure

Event-driven scheduling automatically adjusts the application execution by dynamically incrementing and decrementing the processing frequency of software components. With event-driven scheduling, execution of Units is performed only if events are present at the Units’ input queues. Events represent work to do. Therefore, by default, the application’s execution pressure is automatically driven by the work the application needs to perform.

In DSPE, it is possible to profit from this behaviour to improve the performances and the reactivity of parallel stream-processing applications. By using special-purpose events, it is possible to explicitly drive the application’s execution pressure where helpful or more opportune. In particular, this artifice may be used with the goal of keeping all the present task queues as full as possible.

DSPE supports two strategies for managing the execution pressure of applications:

**Pull-pressure**: with this strategy, when a software component queues a task to the soft co-processor infrastructure, it may also send special-purpose “next” events to explicitly request further stream data from preceding components (see Figure 6.11). These “next” events provoke the execution of components, which will get ahead as compared with a sequential execution.

**Push-pressure**: with this strategy, the events containing the stream data are, instead, free to flow through the software components. The components where streams originate use “auto-next” events to autonomously continue the production of stream events (see Figure 6.12). Other components just ask to pause if they get overloaded, by using dedicated “stop” connections. In other words, as long as there is no overload, the previously introduced “next” events of the pull-pressure strategy are implicit. Components always work ahead compared with a sequential execution. Apart from “auto-next” events, the only event required with the push strategy is used to resume processing, as soon as the overload has been resolved.

The alternative idea of the push-pressure strategy originates from the fact that waiting for

![Diagram of pull pressure strategy](image-url)

**Figure 6.11**: Illustration of the pull pressure strategy.
explicit requests, with potential unnecessary interruptions of the flow of the stream, may degrade performance. Furthermore, push pressure normally requires fewer communication events. Therefore, the push-pressure strategy incurs fewer overheads and may produce better results than the pull. However, its behaviour is less predictable, and the task queues may sometimes shortly exceed limits. As a consequence, care is required when designing and implementing applications because drawbacks may arise in terms of peaks in the memory occupation, peaks in the maximal total latency, and strong alterations in the application response jitter (see Section 6.14). Normally, either a reasonable amount of memory is available and real-time requirements are not excessively binding, or mechanisms for constraining the push pressure are needed.

For this reason, DSPE provides functionalities for limiting the number of used lightweight tasks. For instance, maximal amounts may be specified for each task queue. Furthermore, by means of adjustable thresholds, it is possible to modify the frequency of "stop" signals and "resume" events. Besides stabilizing the pressure, these additional features may also help to avoid unnecessary short stops and resumes, with consequent improvement of the push-pressure behaviour.

The infrastructures for pull and push pressure are generated automatically from the information provided in the DSPE model. Furthermore the generated functionalities integrate with the automatic management features of Section 5.10 and Section 6.10. If automatic features are all enabled and either pull or push pressure strategies are correctly set up, programmatic management in the component's algorithms may be completely avoided.

Depending on the information specified at model level, many variants of the infrastructure for managing the execution pressure may be generated in the source code of Coprocessor Units. Figure 6.13 provides an illustration of a variant, which features pull pressure for both input and output parts of the Unit.

The infrastructure described in Figure 6.13 utilizes the internal variables of Table 6.4. The source code of Listing 6.6 features a simplified version of the operations performed by the execution pressure infrastructure of Figure 6.13.

Figure 6.12: Illustration of the push pressure strategy.
Chapter 6. Soft Coprocessor Infrastructure

Table 6.4: Internal variables of the infrastructure for pull execution pressure.

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sentOutNexts</td>
<td>to keep track of the number of outgoing “next” events that have been sent, but for which the requested data still have not arrived.</td>
</tr>
<tr>
<td>waitingOutNexts</td>
<td>to store the amount of outgoing “next” events that are waiting to be sent, for instance, because the task queue is full.</td>
</tr>
<tr>
<td>numTasks</td>
<td>to keep track of the total amount of tasks waiting in queues, being processed by soft coprocessors, or waiting to be dismissed by the Coprocessor Unit.</td>
</tr>
<tr>
<td>waitingInNexts</td>
<td>to store the amount of incoming “next” events waiting to be satisfied because there are no ready output events.</td>
</tr>
<tr>
<td>numPendingEvents</td>
<td>to keep track of the amount of ready events waiting to be sent in response to an incoming “next” event.</td>
</tr>
</tbody>
</table>

Figure 6.13: Infrastructure used for managing the execution pressure in Coprocessor Units.
Listing 6.6: Parts of source code generated for the pull pressure infrastructure.

1) IF A DATA EVENT HAS BEEN RECEIVED AND
ALL INPUTS ARE READY IN THE TRANSIT QUEUES

// Updates the counters
sentOutNexts --;
waitingOutNext ++;

// Queues the task
queueTask();
umTasks ++;

2) IF A TASK-READYS EVENT HAS BEEN RECEIVED AND
THE TASK IS DISMISSED

// Updates the counters
numTasks --;
umPendingEvents ++;

// If possible sends output events
if (waitingInNexts != 0 && numPendingEvents != 0) {
    waitingInNexts --;
    numPendingEvents --;
    // Signals that output events have to be sent
    signalOutEventsToSend();
}

3) IF AN INPUT NEXT EVENT HAS BEEN RECEIVED

// Updates counter or signals that output events have to be sent
if (numPendingEvents == 0) {
    waitingInNexts ++;
} else {
    numPendingEvents --;
    signalOutEventsToSend();
}

4) AFTER 1), 2) AND 3) AND FOR EVERY OTHER INCOMING EVENT

// Verifies if new data may be requested
inQueueAvailability = inQueueSize - numQueuedElements
- sentOutNexts;
busyAvailability = maxTaskBusy - numTasks;
pendingAvailability = maxPendingEvents - numPendingEvents;
totalAvailability = inQueueSize + maxTaskBusy + maxPendingEvents
- numQueuedElements - sentOutNexts - numTasks
Chapter 6. Soft Coprocessor Infrastructure

46 - numPendingEvents;

47 // Computes the amount of data to request
nextCount = waitingOutNexts;
49 if (pendingAvailability < nextCount) {
50 nextCount = pendingAvailability;
51 }
52 if (busyAvailability < nextCount) {
53 nextCount = busyAvailability;
54 }
56 if (inQueueAvailability < nextCount) {
57 nextCount = inQueueAvailability;
58 }
59 if (totalAvailability < nextCount) {
60 nextCount = totalAvailability;
61 }
62 return nextCount;

96 // Sends all the requests
64 while (nextCount > 0) {
65 sendOutputNext();
66
68 // Updates the counters
69 sentOutNexts ++;
70 waitingOutNexts --;
71 nextCount --;
73 }

74 // Sends all flagged output events
sendSignalledOutEvents();

6.17 Execution Pressure Patterns

For both the pull and push execution pressures, many solutions are possible when connecting the software components with “next” events or with “stop” signals and “resume” events. One of the most typically used patterns is to send these requests to the Unit that directly precedes in the flow of the data stream (as illustrated in Figure 6.11 and Figure 6.12). For this purpose, State Unit Behaviours also provide support for execution pressure management and can be used in combination with Coprocessor Units in the chained components. However, this is not the only possible solution. In many situations, other connection schemes may be more appropriate. For instance, connections that skip some software component are allowed to reduce the overheads of the infrastructure and to increase the reactivity of the responses.

In the possible connection patterns, the following elementary parts are mostly used:
Stream sources: These are the Units where streams originate. Normally, these Units are either Queue Units or Coprocessor Units. At their outputs, these software components may be managed with both pull and push pressure strategies. Contemporaneously, they may have “auto-next” events for autonomously continuing the production of stream events (see Figure 6.12 for an example).

Stream processors: Units that receive one or more input stream events and produce one or more output stream events as a consequence. Normally, these software components are Coprocessor Units, which communicate both with the preceding and the following components in terms of pull and push pressure information.

Stream routers: These are Units that receive one or more stream events and dispatch the events to one or more Units depending on some control logic. These Units are mostly implemented with State Unit Behaviours and may feature input pressure management, output pressure management, both, or none of them, depending on the application requirements.

Stream targets: These are the consumers of the streams. Normally, they only send pressure requests to preceding components.

To allow designing as many connection schemes as possible, the DSPE-generated code provides dedicated infrastructure for managing more than one incoming pressure management connection. For this purpose, use of Group Gates of next events, stop signals, and release events are allowed.

6.18 Event-Driven Delay Line Support

As introduced in Section 4.5, frequently, state-full components are performance restrainers in parallel stream-processing applications because they represent bottlenecks. As a consequence, if it is possible to remove some data dependencies and transform state-full components into state-less ones, additional performance improvements may be obtained.

The transformation from a state-full component into a state-less one is not always possible or may introduce overheads and side effects. Nonetheless, there are situations in which components, such as FIR filters, have—in principle—a stateless nature, but, for the sake of simplicity, are implemented with internal states in serial applications. When parallel processing with the soft coprocessor infrastructure of DSPE, these internal states are unnecessary obstacles and have to be removed. On the other hand, there are situations in which real state-full components, such as IIR filters or components with feedbacks, are present. Even in these situations, in parallel stream-processing applications, a partial reduction of the data dependency is sometimes possible. This depends on the possibility of tolerating side effects in the algorithms as explained below.

The event-driven delay lines of DSPE facilitate and automate the transformation of state-full components into state-less ones. In principle, the available solution allows configuring the
transit and arm queues of the event-driven infrastructure to store a sufficient amount of old data, which may be used as a replacement of the internal delay lines of the serial application versions. This is done by holding the events in transit or in arm queues (see Section 5.7) for more processing rounds than normal, before being released. Furthermore, the provided infrastructure also integrates with the soft coprocessor infrastructure when parallel processing. As introduced in Section 4.11 and Section 6.6, the information contained in the input events received by Coprocessor Units is read-only and, therefore, it may be accessed simultaneously by lightweight tasks. By also providing task-level access to the data stored in the input event-driven delay lines, the permanent states (see Section 6.9) required for the internal delay lines can be removed and the Coprocessor Unit transformed into a state-less one.

Concerning the output Event Gates, the solution is not as straightforward. Output Event Gates are not read-only and, therefore, parallel processing is not allowed even if output event-driven delay lines are used. Additional artifices are required for parallel processing. If the amount of data stored in the delay line is reasonably smaller than the amount of data processed by the tasks, and if the algorithm tolerates small side effects, it is possible to explicitly enable the concurrent execution of tasks. Then, the Coprocessor Unit Behaviours may do the reconciliation of the portions of data at the beginning and end of the processed parts, for instance, by reprocessing some values or by performing interpolation.

### 6.19 Thread Management Done by the Coprocessor Manager

A wide range of threading and synchronization libraries written in C/C++ is available for multi-cores. DSPE supports PThreads and WinThreads. At the DSPE model level, it is possible to select which version has to be used. Furthermore, the generated source code provides customization points, in the thread manager, for integrating APIs and frameworks not directly supported by DSPE (see Section 3.23 for more information).

The soft coprocessor infrastructure of DSPE utilizes and synchronizes threads by using the thread management infrastructure made available by the thread manager (see Section 3.24). Each soft coprocessor has an assigned thread. Normally, all coprocessor threads have the same priority. Furthermore, soft coprocessors take advantage of the immutable nature of events and tasks to minimize synchronization costs (see Section 4.11 and Section 6.6 for more details). The source code generated by DSPE uses synchronization primitives only at crucial hotspots. To avoid unnecessary context switches, spinlocks and atomic actions are used in the coprocessor manager (see Section 4.10). Furthermore, particular care has been taken to reduce the size of the atomic sections in the source code and to eliminate any contained loop having non-predictable execution time.
Concerning thread management, two versions of the coprocessor manager may be generated:

- With *busy-waiting* coprocessors, coprocessor threads never go to sleep and spin idly if task queues are empty.
- With *sleeping* coprocessors, coprocessor threads are allowed to go to sleep if they are unable to find tasks to process.

Busy-waiting is more efficient, but processing resources may be wasted if the application execution pressure is insufficient. Sleeping is fairer, but introduces some overheads when waking-up and sending the threads to sleep. Therefore, it may be less reactive and incur some performance reductions. For the application versions using an independent thread for the scheduler (e.g., the scheduler contribution version), similar solutions are available for managing the scheduler thread. Section 5.14 provides more information about scheduler suspensions and resumes.

Another important functionality featured by the coprocessor manager allows automatically assigning thread affinities to soft coprocessors. When affinities are assigned, each coprocessor thread executes on a predetermined processor core. Consequently, the application execution is less prone to context switches and may result more stable. In DSPE, thread affinities are assigned at initialization time. Thread affinities are particularly effective in parallel stream-processing applications with high and stable execution pressure.
Today, there are many types of hardware architectures for parallel processing. The most frequently used are multi-core and many-core processors, which share some similitude because they are all chip-level multiprocessors. However, as well as a different number of cores, these processors frequently feature dissimilar hardware components like control units, caches, and communication buses. As a consequence, multiprocessors that are more versatile and provide valuable performance to most types of applications are used as general-purpose solutions, whereas other processors more effective for specific computations are utilized as special-purpose solutions.

In addition, heterogeneous hardware platforms are also frequently used. These platforms combine parallel processors of different types with the goal of simultaneously obtaining the benefits of both general-purpose and special-purpose solutions. Heterogeneous platforms may, for instance, be obtained by combining multi-core CPUs and many-core GPUs or by using CPUs in combination with FPGAs. General-purpose processors are normally used as masters for performing control operations, whereas special-purpose processors are used as slave accelerators for performing specific computations. Depending on the type of integrated processors, these platforms are used in a range of specific contexts from high-performance to embedded computing.

The DSPE soft coprocessor infrastructure mainly features functionalities for general-purpose multi-cores (see Chapter 6). However, the concept at the basis of the soft coprocessor infrastructure is similar to the concept of the accelerators, which may be considered hard coprocessors in heterogeneous hardware platforms. Therefore, with the goal of supporting parallel stream processing on heterogeneous combinations of multiprocessors, DSPE has been enhanced to exploit accelerators for the execution of the generated applications. The accelerator infrastructure described in this chapter integrates functionalities within the DSPE soft coprocessor infrastructure for application execution on many-core hardware accelerators. Even if many parts of the accelerator infrastructure are useful for any type of accelerator, the current version of the DSPE accelerator specializes to the utilization of GPU accelerators. As a consequence, this chapter focuses attention on GPUs as accelerators.
Chapter 7. Accelerator Infrastructure

Today, GPUs are massively parallel processors featuring strong computational power. Initially, GPUs were used only for performing special-purpose processing before displaying the images on video. However, with CUDA [31, 73, 94, 74], OpenCL [106, 135, 128], and Microsoft DirectCompute, it is now possible to execute any type of algorithm on the GPU and to bring the results back on the CPU.

The DSPE accelerator infrastructure allows integrating GPU kernels written in C for CUDA [32] within the DSPE event-driven soft coprocessor architecture. The DSPE model and generators have been extended with support for scheduling GPU kernels and for transferring data streams between CPUs and GPUs, and backwards.

This chapter is the last chapter of this dissertation that describes the software architecture of the source code generated by DSPE. The accelerator infrastructure is a complement to the event-driven and soft coprocessor infrastructures, which have been presented in previous chapters. Instead, the next chapter presents the DSPE model, which also features, as part of its functionalities, specific support for configuring the accelerator infrastructure.

7.1 Stream Processing with Multi-Cores and GPUs

GPUs allow for the execution of special parts of the program called kernels, executed hundreds of times in parallel on small portions of the data to process. Because of their hardware architecture, GPUs are very effective when performing identical computations on all the GPU cores, whereas the produced performances quickly degrade as soon as algorithms execute branch operations (for instance, as consequence of "if" instructions). For this reason and because of the amount of featured cores, fine-grained data decomposition is the approach for parallelism mostly used with GPUs. As mentioned in Section 4.1, depending on the type of performed computation, this form of parallelism may not be easy to exploit. It is mostly effective if many data of the same type and with few data dependencies have to be processed in the same way. Consequently, GPU kernels are mostly used for parallel calculations on matrixes and arrays of data, such as in linear algebra and digital filtering.

Figure 7.1: Combined use of CPUs and GPUs.
In stream-processing applications, GPUs mostly execute small but time-consuming parts of the source code. It is possible to take advantage of heterogeneous combinations of multi-core CPUs and many-core GPUs, by exploiting coarse-grained decompositions on the CPUs and fine-grained data decompositions on the GPUs. CPU cores are mainly responsible for scheduling the execution of GPU kernels but, depending on the nature of the application and the amount of work performed on the GPU, may also be used for executing processing algorithms (see Figure 7.1).

A GPU is fully employed by a kernel, if each of its cores has sufficient data to process. To hide the latency that may be caused by register dependencies, GPUs uses an execution model that asks for a large number of simultaneously active threads. In this model, all the threads that simultaneously perform the kernel execution have to be subdivided in a grid of blocks of threads (see Figure 7.2). As explained in the CUDA documentation and literature [32, 33, 74], with the GPUs available today, each CUDA block should at least be composed of 128 threads. Furthermore, the number of CUDA blocks in a CUDA grid should also be sufficiently large to keep the hardware busy. These numbers may grow for future GPU generations.

To keep all these threads occupied in a stream-processing application, a large number of contemporaneously available stream items may be required. On the other hand, when fully exploiting heterogeneous hardware, buffering of stream items may also be needed for parallel
processing on multi-cores. This situation may lead to problems in many types of real-time stream processing application, in particular, if time-critical requirements have to be satisfied.

There are further performance costs related to the use of GPUs for real-time stream processing. These costs are mainly caused by the data transfers between CPUs and GPUs and are justified only if the speed-up achieved by the algorithms on the GPUs is sufficiently strong. Again, low-latency requirements may significantly limit the amount of simultaneously available data to process and compromise performance. In this context, possibilities to overlap the memory transfers and the algorithm execution may help to reduce the latency. For instance, CUDA streams allow for simultaneously executing kernels and transfer data for non-overlapping data portions of the streams. If the kernel execution time is longer than the transfer times and if the GPU is constantly under pressure, this solution allows absorbing part of the transfer overhead. The efficient execution of GPU kernels may face additional obstacles. For example, performance may be limited by decreased availability of registers and shared memory or by non-coalesced access to memory. The CUDA documentation and literature [32, 33, 74] provide more insight on these topics (see also Section 7.12 for some details).

### 7.2 Design Guidelines for Stream Processing with Heterogeneous Processors

This Section provides important guidelines used in many parallel stream-processing applications designed with DSPE with the goal of targeting heterogeneous combinations of CPUs and GPUs. Information about these guidelines is important to allow understanding the software architecture of these types of applications.

1. The most time-consuming software components are executed on the fastest processing resources. Normally, these are the GPUs. If kept under constant pressure, these types of accelerators may provide significant performance improvements.

2. Among the time-consuming components, priority on the accelerators is given to components featuring data parallelism with few data dependencies. In stream-processing applications, these types of components allow for the maximum profit from fine-grained data decompositions.

3. From the point of view of the application design, care is taken to put more workload on the fastest processing resources than on slower resources. If GPUs are initially loaded with more processing tasks than CPUs, then, at run time, the source code generated by DSPE allows balancing the total resource load by performing heterogeneous work stealing (see Section 7.9 for more information).

4. As a consequence of the preceding guidelines and due to the execution model and specificity of GPUs, CPUs are used for executing the GUI, control and scheduling parts of the application.
5. If idle CPU cores are still available, it is possible to perform heterogeneous processing by executing software components on the CPU too. This may be done either directly as part of the scheduler or by means of soft coprocessors (see Section 6.7 for more information). As an alternative, heterogeneous work stealing may be used for this purpose (see Section 7.9 for more information).

7.3 Integration of the Accelerator and the Soft Coprocessor Infrastructure

The DSPE abstract model and the DSPE generators allow for the production of the accelerator infrastructure, which features functionality for automatic kernel execution and for automatic data transfer between CPUs and GPUs. The available support allows management of all the CPU-GPU interactions directly from the lifecycle phases of software components or from the lifecycle phases of lightweight tasks. To distinguish between the different types of tasks, all lightweight tasks that manage GPU kernels are called kernel tasks in DSPE. Compared with the solution that directly uses components for managing kernels, the solution using kernel tasks (see Figure 7.3) frequently is more effective because it offers more possibilities in terms of combinations of coarse-grained decompositions and fine-grained decompositions. Furthermore,

Figure 7.3: Integration of the accelerator and the soft coprocessor infrastructure. Operations are performed as follows: 1. events are exchanged between Software Units; 2. when events are received by Units featuring GP Unit Behaviours, kernel tasks are prepared and queued; 3. kernel tasks are processed by soft coprocessors by performing the required data transfers and launching the kernel execution; 4. kernels are executed and results are transferred back to the CPU; 5. task-ready events are sent to the Units.
kernel tasks support the automatic integration of CUDA streams for the execution of tasks (see Section 7.7) and allow heterogeneous work stealing (see Section 7.9). As a consequence, with kernel tasks, it is easier to absorb the overhead introduced by the data transfers and to obtain load balancing on both the CPUs and the GPUs.

### 7.4 GP Unit Behaviours

Most of the accelerator infrastructure is provided by GP Unit Behaviours, which are special-purpose Unit Behaviours that extend and specialize the other types of Unit Behaviours. There are two types of GP Unit Behaviours: GP Implementations and GP Block Optimizations. As a main enhancement, GP Unit Behaviours introduce additional lifecycle phases (see Figure 7.4) specific for managing the execution of GPU kernels and the needed data transfers (For more information about lifecycle phases see Section 3.15, Section 5.11, and Section 6.8). Figure 7.4 shows the lifecycle phases of a GP Unit Behaviour integrated with the functionalities of a Coprocessor Unit Behaviour (see Section 6.8). The integration is similar, when done with the functionality of a simple Unit Behaviour or a State Unit Behaviour (see Section 3.16 and Section 5.9).

The most prominent difference is that the process phase and the associate initialize parameters and finalize parameters phases (represented with “Init” and “Fin” in Figure 7.4) encapsulate the GPU kernel and are executed on the available GPUs. As a consequence, the additional lifecycle phases of Table 7.1 are introduced and, for all types of GP Implementations, pre and post phases of the initialize memory, copy to device, copy from device, and dispose memory phases are also provided. These phases, which are represented in Figure 7.4 with small rectangles, have to be used for performing computations on parameter values when block processing is active. Pre and post phases are executed only once for each block of the stream data, as explained in more detail in Section 3.19.

<table>
<thead>
<tr>
<th>Lifecycle Phase</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initialize memory</td>
<td>this lifecycle phase is used for allocating the required memory regions on the GPU global memory.</td>
</tr>
<tr>
<td>Copy to device</td>
<td>a lifecycle phase that may be used for copying the stream data to the GPU global memory.</td>
</tr>
<tr>
<td>Kernel execution</td>
<td>with this phase, it is possible to start the kernel execution.</td>
</tr>
<tr>
<td>Copy from device</td>
<td>this lifecycle phase has to be used for copying all the results of computation from the GPU global memory back to the CPU memory.</td>
</tr>
<tr>
<td>Dispose memory</td>
<td>this phase is useful for releasing the memory regions previously allocated on the GPU.</td>
</tr>
</tbody>
</table>

Table 7.1: Additional lifecycle phases of GP Unit Behaviours.
7.5 Automatic Support for Kernels

In a way similar to the automatic supports for events and tasks (see Section 5.10 and Section 6.10), the DSPE generators also provide automatic functionalities for managing kernels. These automatic features are mostly profitable if block processing is enabled (see Section 3.19). By means of the lifecycle phases of GP Unit Behaviours (see the preceding Section), DSPE users may manage the execution of kernels and the required data transfers programmatically. However, in many situations, they are not forced to do so. The source code generated by DSPE is able to allocate the GPU memory and copy the blocks of stream data to the GP automatically. Similarly, source code may be generated for automatically executing kernels, for copying results back to the CPU, and for releasing the previously allocated GPU memory. When these
functionalities are enabled, GP Unit Behaviours also automatically separate the available stream data in CUDA grids and CUDA blocks to allow efficient use of the GPUs. The approach used for separating the data into CUDA blocks can be configured at the DSPE model level. Listing 7.1 provides an example of the generated source code.

Listing 7.1: Example of the source code generated for kernel execution.

```c
// Memory initialization performed
// for every data gate of the software component
unsigned int memorySize = BLOCK_SIZE * sizeof(float);
cudaMalloc((void**) &gpu_input, memorySize);
cudaMalloc((void**) &gpu_output, memorySize);

// Copy to device performed
// for every input data gate
cudaMemcpyAsync(gpu_input, component_input, memorySize, cudaMemcpyHostToDevice, stream);

// Execution of the kernel
dim3 threads(NUMBER_OF_THREADS);
dim3 grid(BLOCK_SIZE / threads.x);
processingKernel<<<grid, threads, 0, stream>>>
(gpu_input, gpu_output);

// Copy from device performed
// for every output data gate
cudaMemcpyAsync(component_output, gpu_output, memorySize, cudaMemcpyDeviceToHost, stream);

// Memory clean-up
cudaFree(gpu_input);
cudaFree(gpu_output);
```

7.6 Special-Purpose Support for Kernel Tasks

To support different GPU hardware architectures and stream-processing approaches, DSPE allows specifying how soft coprocessors have to manage kernel tasks. The following solutions are available:

**Locking coprocessors**: as soon as a soft coprocessor operates a queue of kernel tasks, it locks the queue. Afterwards, it will be the only soft coprocessor able to schedule the execution of kernel tasks from that queue. This version may minimize GPU memory consumption and reduce the number of GPU context switches. However, it may produce limited speed-up results because kernel tasks are consumed sequentially as if they always were state-full tasks.
This approach is more effective, if used in combination with CUDA streams, which allow making the overhead of data transfers between CPUs and GPUs unnoticeable (see the next section for more details).

**Sharing coprocessors:** in this version, all soft coprocessors are allowed to run kernels on GPUs by processing the kernel tasks produced by GP Unit Behaviours. Performance may significantly increase because kernel tasks can be consumed by the GPUs in parallel. However, GPU hardware that allows concurrent execution of kernels is required. This type of support is not always present, in particular in small and non-recent GPUs. Furthermore, with this approach, GPU memory utilization may significantly increase because of the required contemporaneous presence of more than one kernel context.

### 7.7 Integration of CUDA Streams

DSPE features specific support for automatically managing CUDA streams [32]. This functionality allows hiding the overhead of the data transfers between CPUs and GPUs behind the time spent by the GPUs for executing kernels. For each consumed kernel task, the source code generated by DSPE uses an individual CUDA stream (as shown in Figure 7.5). Consequently, kernel processing and data transfers between CPUs and GPUs may overlap.

When CUDA streams are used, the DSPE-generated code features scheduling of kernel task that is completely asynchronous (see Figure 7.6). Soft coprocessors—instead of waiting that the kernel execution and data transfers of a preceding kernel tasks is complete—are able to launch the execution of any number of CUDA streams at any time. Then, it is up to the CUDA execution environment to schedule the execution of the CUDA streams. A special-purpose list of streams is stored by soft coprocessors for keeping track of all CUDA streams under processing. Soft coprocessors regularly update this list and verify if the execution of the CUDA streams is complete. When complete, they return the associated kernel tasks to the GP Unit Behaviour that originated the task, as done for normal lightweight tasks (see Section 6.4). To allow reducing the memory requirements of the simultaneously processed kernel tasks and to limit wait times, in the DSPE model it is possible to configure the maximum number of simultaneously used CUDA streams.

![Figure 7.5: Example of the interleaving performed when using CUDA streams](image-url)
Chapter 7. Accelerator Infrastructure

Figure 7.6: With CUDA streams, the scheduling of kernel tasks is completely asynchronous. In the diagram, s1, s2 and s3 are streams.

7.8 Memory Management: GP States and GP State Pools

When performing streams processing on heterogeneous combinations of parallel processors, there are two approaches possible for managing the memory regions required by the kernels in the global memory of the GPUs. If the size of these memory regions frequently changes, allocation and disposal of memory before and after each kernel execution is usually required. Instead, if these memory regions always remain the same size, it is possible to allocate memory the first time these regions are needed and to recycle the memory regions subsequently. For this purpose, DSPE features GP states, which are special-purpose C structs containing the references to the memory regions allocated on the global memory of the GPUs. In a way similar to the support provided for events and lightweight tasks (see Section 5.4 and Section 6.5), a pool of GP states is maintained to allow reutilizing the GP states. As long as a GP state is not available in the pool, a new one is created by the generated infrastructure, and the memory regions required in the global memory of the GPU are allocated. Otherwise, if a GP state is available in the pool, the contained references to the memory regions are reused, and the memory allocation costs are saved.

The pool of GP states may be used in combination with the sharing coprocessor version (see Section 7.6) or if CUDA streams are used (see the preceding section). In these situations, more than one memory region of the same type can be allocated on the GPU at the same time. In other situations, only one GP state is maintained and reused. In addition, as part of the GP state, it is also possible to recycle CUDA streams. Instead of allocating and disposing the CUDA streams for each kernel execution, references to the Streams are maintained and reused.
7.9 Heterogeneous Work Stealing

With the purpose of efficiently processing streams on heterogeneous hardware, DSPE supports heterogeneous work stealing of kernel tasks as a further improvement for load balancing. The provided solution allows stealing kernel tasks that are produced by state-less software components. If heterogeneous work stealing is enabled, both a GPU kernel and a standard C implementation of the algorithm have to be specified for the software components. As a consequence, components become hybrid by being able to execute both on the GPU and on the CPU. In an application featuring heterogeneous work stealing, either work sharing or work stealing (see Section 6.12) may be used for executing normal lightweight tasks, kernel tasks, and hybrid tasks. However, if a soft coprocessor owns a queue of hybrid tasks and executes them on a GPU, idle soft coprocessors may steal tasks heterogeneously from the back of the queue and execute the stolen tasks on the CPU instead of on the GPU (see Figure 7.7).

If the workload on the GPU exceeds the workload on the CPU, this approach brings benefits because it maximizes the CPU load by partially unloading the GPU. However, it is important to steal tasks from the back of the queue because hybrid tasks normally execute significantly faster on the GPU than on the CPU. If tasks would be stolen from the head of the queue, to keep the data stream in the correct order, an excess of processed tasks waiting at the output of Units could appear. These waiting tasks may compromise the application execution pressure.

If, in agreement with Section 7.2, the most powerful resources are the stealing victims, heterogeneous work stealing is effective compared with other partitioning approaches because it favours the execution of tasks on fast resources, as long as these resources are available. If slow processing resources execute the OS, GUI and remaining application parts, slow processing resources are better exploited in the role of the stealers. They may help to accelerate the execution only when idle, with consequent minimal impact on the application responsiveness.

![Figure 7.7: Heterogeneous work stealing performed by the accelerator infrastructure](image-url)
The performance improvements resulting from heterogeneous work stealing may depend on many factors. In most situations, it helps to maximize the performance. However, the amount of available idle CPU cores has a strong influence on the results. If most of the processor cores are already performing other processing operations, performance improvements from heterogeneous work stealing may be nullified. Similarly, if the algorithm executes significantly faster on the GPU than on the CPU, the additional acceleration introduced by heterogeneous work stealing may be only partially relevant. For example, if the GPU processes 100 hybrid tasks, while, at the same time, CPU cores are able to process 1 of these tasks heterogeneously, the additional improvement is 1%. Furthermore, to be efficient, heterogeneous work stealing requires a correctly designed software architecture (see Section 7.2) and strong execution pressure on the GPUs (see Section 7.11).

7.10 Support for More Than One Accelerator

The source code generated by DSPE allows for utilizing more than one GPU accelerator from the same stream-processing application. The provided accelerator infrastructure autonomously distributes the execution of kernels to the available resources. As a consequence, it is possible to perform parallel stream processing on various combinations of CPUs and GPUs.

However, in stream-processing applications, finding an application design that profits from more than one GPU at the same time may be difficult. As introduced in Section 4.5, data dependencies, bottleneck components and other types of bottlenecks limit parallelism and may nullify the benefits of additional accelerators. Furthermore, with more than one GPU, the application latency may further increase with consequences on the real-time behaviour of the applications. In particular, with more than one GPU, it is difficult to keep all the accelerators constantly under pressure. More details about this topic are provided in the next section.

7.11 GPUs and the Application Execution Pressure

As introduced in Section 4.5, in most stream-processing applications slow software components are the bottlenecks. GPUs allow significantly accelerating slow software components, with consequent potential alleviation of the impact they have on the application execution pressure. In other words, by carefully selecting the software components that execute on the GPUs, it is possible to improve the global application execution pressure and obtain performance improvements.

However, as explained in Section 7.1, to be efficient, GPUs are the first that need to be kept under high execution pressure and, consequently, more data have to be buffered than in stream-processing applications executing only on multi-cores. As discussed in Section 1.2, in real-time stream-processing applications, this fact may be problematic considering the real-time constraints. The infrastructure provided by DSPE for pull and push execution...
7.12 Performance Considerations About GPU Kernels

The speed-ups obtained with GPUs when accelerating specific algorithms of the DSPE-designed applications strongly depend on how much faster GPU kernels are than their standard C implementations for CPUs. With the DSPE infrastructure for profiling (see Section 3.25), it is possible to measure and compute the acceleration obtained by the algorithms on the GPU. Measures of the GPU kernel speed may be taken both on the GPU side and on the CPU side. GPU-side measures only pertain to kernel execution costs, whereas CPU-side measures may also include memory allocation costs, data transfer costs, and kernel launch costs. With all these measures and with a measure of the speed of a standard C implementation of the algorithm, it is possible to compute the pure algorithm acceleration and a version of the acceleration that includes all overhead. Then, the performance improvements brought by kernels on the complete application may be estimated.

If the pure acceleration of the algorithm is strong, the resulting overall speed-up of the application should also be significant. However, in stream-processing applications the costs required for managing memory and transferring the data may be high and, therefore, the impact of the overhead has to be taken into account. Use of GP states (see Section 7.8) may be helpful for eliminating the costs of memory allocation and disposal. Additionally, the use of CUDA streams (see Section 7.7), which should be disabled when performing the measures, may help to absorb the data transfer costs.

Many factors influence pure algorithm speedup. Some of them are

1. **The type and granularity of data decompositions** allowed by the algorithms executed on the GPU. Fine-grained data decompositions have to be used on GPUs; therefore, algorithms that decompose the data in many small parts and with few data dependencies are strongly favoured.

2. **The type of computation performed on the GPUs.** Many GPU hardware architectures support floating-point computations poorly, in particular, in terms of double precision operations.

3. **The structure of the source code executed on the GPUs.** GPUs are more effective when running algorithms that contain only a few branch instructions (due, for instance, to “if” instructions). As soon as a branch instruction is executed, part of the parallel execution on the GPU cores may be blocked, and sequential execution may instead be performed.

4. **The utilization of the shared and constant memories available on the GPUs.** These types of memories are significantly faster to access than global memory. Therefore, they
should be exploited as much as possible. In particular, concerning the shared memory, performances may be strong if the ratio between read operations and write operations is high and if the parts of the shared memory written by one thread are read by many other threads. However, shared and constant memories are normally small, and, consequently, their efficient exploitation is difficult.

5. The utilization done by the algorithms of the available processors registers. GPU registers are a precious resource that may significantly limit performance if they are overused. To understand the impact of the register utilization on GPU kernel performance, the CUDA compiler allows for calculating a value called "occupancy" (see the CUDA documentation [32, 33] for more information).

6. The size of the CUDA grid and the size of the CUDA block (see Section 7.1). If these values are excessively small, the execution pressure on the GPU may be insufficient and the resulting performance may be poor. The ideal sizes mostly depend on the GPU hardware architecture used. The CUDA documentation [32, 33] provides more information about how these values can be configured correctly.

There are many other factors that influence results, such as coalesced access to memory and use of GPU optimized functions. More information about how GPU kernels may be optimally designed and implemented is available in the CUDA documentation [32, 33].
8 DSPE Model

The DSPE model is the most important part of DSPE. All the DSPE functionalities gravitate around the model, which is used for storing the abstract information of software components, software architectures, and other features of parallel stream-processing applications. The DSPE model is used directly during development and is accessed by means of wizards, editors, and assisting functionalities, such as refactoring and model transformations (see Chapter 9 for more information). The abstract information, which results from model-driven development, describes the structure and behaviour of the applications at a high level. Then, the DSPE generators query the DSPE model and use the contained information for generating the source code described in Chapter 3, Chapter 5, Chapter 6, and Chapter 7.

The development of a DSL based on a domain-specific model and the development of its generators is similar to the development of a general-purpose programming language, such as C or Java, and its compilers. In the first place, syntax and semantics of the DSL have to be defined. Then, generators need to be implemented. Generators play a role similar to compilers in general-purpose programming languages. Therefore, The DSPE DSL shares some similarity with a general-purpose programming language. It features a type system, support for modular development and functionalities for describing the behaviour of programs. However, the DSPE DSL does not provide direct support for programming algorithms like loops and conditional instructions. Different from general-purpose programming languages, the DSPE DSL is used for combining software components and for configuring the structure and behaviour of specific features of stream-processing applications.

As introduced in Section 2.6, DSLs may have one major drawback, which is a consequence of their high abstraction level. If the DSL is not sufficiently flexible, the resulting producible application family may be small, and fine-tuning may not be possible. In these situations, direct modification of the application source code provides a solution. However, this approach may prevent continued use of the DSL, with consequent interruption of the rapid prototyping and customization process. To deal with these kinds of problems, DSPE provides a hybrid extensible solution, which allows using the DSL in combination with general-purpose programming languages. With this solution, it is possible to specialise the DSPE DSL functionality...
Chapter 8. DSPE Model

for fitting domain-specific needs.

On the other hand, the DSPE DSL may be considered an expansion of the syntax and semantics of C and C++. For instance, the DSPE type system allows customizing standard C types with user-defined constraints. These constraints are put into effect in the generated source code. The DSPE DSL is abstract and high level. However, coexistence and cooperation with C and C++ permits application fine-tuning and optimization. To enhance this synergy between languages, the DSPE development environment provides special-purpose functionalities, such as extractors and integration with the CDT Eclipse plug-in [43]. More information is provided in Section 9.9.

8.1 EMF

The DSPE model and the DSPE generators have been built with the Eclipse infrastructure for model-driven development and generative programming. This section introduces the Eclipse Modelling Framework (EMF) [44, 126], a mature framework that allows creating structured data models in Java. The DSPE model is an EMF instance. More information about the development of the DSPE generators is provided in Section 9.5.

Essentially, EMF allows producing Java classes for managing models and run-time support for viewing and editing the information contained in these models. For example, EMF features functionalities for the automatic serialization of model elements and support for modifying the model contents with transactions. EMF stores the model information in the XML Metadata Interchange format (XMI) [146], a standard for metadata information defined by the Object Management Group (OMG) and based on the Extensible Markup Language (XML) [13]. A simplified example of the EMF persistence (used for a Coprocessor Scheduler) is provided in Listing 8.1.

Listing 8.1: Simplified example of the XMI persistence for a Coprocessor Scheduler

```xml
<?xml version="1.0" encoding="UTF-8"?>
<xmi:XMI xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI"
 xmlns:dspe="http://www.systemdesigner.ch/dspe"
 <CoprocSchedulerDefinition>
 <allUnits xmi:type="UnitDeclaration" name="MatrixS">
 <unitDefinition xmi:type="SoftwareUnitDefinition"
 href="/SoftwareUnits/MatrixSource"/>
 </allUnits>
 <allUnits xmi:type="UnitDeclaration" name="MatrixM">
 <unitDefinition xmi:type="SoftwareUnitDefinition"
 href="/SoftwareUnits/MatrixMultiplier"/>
 </allUnits>
 <allUnits xmi:type="UnitDeclaration" name="MatrixT">
 <unitDefinition xmi:type="SoftwareUnitDefinition"
 href="/SoftwareUnits/MatrixTarget"/>
 </allUnits>
</CoprocSchedulerDefinition>
```
To facilitate the development of the DSPE generators and other functionalities provided by the DSPE environment, DSPE features additional infrastructure specific for the DSPE model, as a complement to the infrastructure featured by EMF. For instance, DSPE provides many special-purpose features for querying the DSPE model and for managing it in an aspect-oriented way. This infrastructure and additional DSPE model features, such as domain-specific validation rules, are described in Chapter 9. The rest of this chapter describes syntax and semantics of the DSPE model.

### 8.2 Base Model

In EMF, all model objects extend the common parent EObject, which provides common functionality, such as for accessing the model object content and for navigating to the object container and its root EMF Resource (more details about Resources are provide below).

Similarly, the DSPE model provides abstract parents too, which collect features common to all DSPE model objects. All DSPE abstract model types are part of an independent model layer called base model (see Figure 8.1). The root of all DSPE model objects is the Base Element, which owns two direct child elements: Entity and Primitive Element.

Each Entity is an EObject, which, by means of Primitive Elements, is used to store information or aggregate other Entities. An Entity has 4 main specializations:

- **Definition**: in DSPE, all root model Entities (not contained in other Entities) extend the abstract model type Definition, which provides functionalities common to all Definitions. Each root model entity is stored in its own file and is identified by a name, which matches the name of the file. Furthermore, each Definition has a specific editor in the DSPE environment (see Chapter 9 for more information).
• Declaration: model Entities contained in Definitions may be either Declarations or Sub Entities. Declarations are collected in groups and are special-purpose Entities owning a name and a reference to a Definition. The name has to be unique inside the containing group. Conceptually speaking, compared to object-oriented programming, the relationship between a Definition and its Declarations is similar to the relationship existing between a class and its objects. Definitions may be used for specifying types of model elements and Declarations may be used for declaring their instances.

• Sub Entity: like Declarations, Sub Entities are also contained in Definitions and collected in groups. However Sub Entities have no particular behaviour like Declarations and are primarily used to aggregate information or build recursive model structures.

• Aspect: to provide infrastructure for aspect-oriented domain modelling (see Section 2.4), the base model contains the abstract model type Aspect. In DSPE, an Aspect is similar to a Definition. It is a root model entity. However, it has no name. An Aspect is associated in a one-to-one relationship with a Definition and is stored in the same file of the Definition (see Figure 8.2). The base model provides dedicated functionality for Aspects, such as aspect-oriented model traversal. For information about DSPE Aspects see Section 8.6.
In DSPE, each Definition and its associated Aspects are stored in an EMF Resource, which is saved in an individual file. All DSPE model files are collected in Systems, which are in DSPE the equivalent of Eclipse projects. At the OS level, Systems have an associate hierarchy of folders, used for organising the contained model files. Furthermore, Systems and Definitions extend the abstract model type Primary Source, which is used as a common parent for all model resources for which it is possible to independently start generation. Thereby, the DSPE development environment activates buttons and actions in context menus for executing generators on Definitions and Systems.

The Primitive Elements of the base model are: Fields, Relations, Handles, and Groups. They are used for editing and storing different types of information in model Entities:

- **Fields** are used to manage textual information like strings and numbers.

- **Relations** are used to reference Definitions. These references are stored as paths to the Definitions in the file system.

- **Handles** are used to reference Declarations and Sub Entities. Each Handle allows the selection of a model Entity in a Group of Declarations or Sub Entities. The reference of a Handle store the complete path to the Declaration or Sub Entity, including the path to the containing Definition. In DSPE, simple Handles may be combined in linked Handles and aggregate Handles. These compound types of Handles allow advanced navigation and filtering of Declaration references.

- **Groups** are used to collect Fields, Relations, Handles, Declarations and Sub Entities of the same type.

Figure 8.2: Contents of an EMF Resource in DSPE.
All types of Entities may contain either Primitive Elements, Groups of Primitive Elements, or Groups of Declarations and Sub Entities (see Figure 8.3). This solution allows building complex aggregates of DSPE-model elements for describing highly structured information.

Mixins are also important model objects featured by the base model. Mixins are EObjects used to specify partial Entities. Mixins may be used for each type of entity as additional parents. Therefore, Mixins allow multiple inheritances at the model level. By taking advantage of Mixins, it is possible to reuse the model functionality and profit from advanced forms of model-level polymorphism. The DSPE model extensively uses Mixins (see Annex A for some examples).
8.3 DSPE Model Entities

The DSPE model is composed of DSPE model Entities, which extend and specialize the abstract model types provided by the base model. In DSPE, structure and behaviour of real-time stream-processing applications are described, in a domain-specific way, by instantiating, editing, and combining these DSPE model Entities. Editors and wizards (see Chapter 9) provide the visual surface for managing the various model Entities.

The following sections provide an overview of the DSPE model. Annex A contains the complete interface of the DSPE model represented in UML.

8.4 DSPE Sub-Models

The DSPE model has been subdivided into sub-models to facilitate its maintenance and growth:

**Ground model:** provides the most simple and fundamental elements of DSPE. With the ground model, it is possible to specify simple real-time stream-processing applications, which have generated source code that mostly uses the base software architecture. Some support for the event-driven infrastructure is also included in the ground model.

**XT model:** provides support for Composites, for the entire soft coprocessor infrastructure, and for the advanced features of both the event-driven infrastructure and the DSPE extensible type system.

**GP model:** provides functionalities for specifying and generating features of the accelerator infrastructure. The available functionalities include support for integrating GPU kernels written in C for CUDA.

**GUI model:** contains all model elements required for designing the graphical user interfaces, from which it is possible to generate source code parts for driving the execution of applications.

Each sub-model is an Eclipse plugin, which may be activated or deactivated depending on user needs. The ground model may be used alone. The XT model and the GUI model depend on the ground model. The GP model depends on the ground model and the XT model.
8.5 DSPE Model Definitions

The DSPE model Definitions may be grouped by functionality as shown in Table 8.1

Table 8.1: List of Definitions of the DSPE model

<table>
<thead>
<tr>
<th>Base support for stream processing</th>
<th>Standard Gate Definition, String Gate Definition, Software Unit Definition, Configuration Definition, Sequence Definition, C Implementation Definition, C Block Optimization Definition, Application Definition, Runner Behaviour Definition, Requirement Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Support for composition</td>
<td>Composite Definition, Wrap Implementation Definition, Wrap Block Optimization Definition</td>
</tr>
<tr>
<td>Support for event-driven execution</td>
<td>Signal Gate Definition, Event Gate Definition, String Message Gate Definition, State Implementation Definition, State Block Optimization Definition, Scheduler Definition</td>
</tr>
<tr>
<td>Advanced support for the extendible type system</td>
<td>Pointer Gate Definition, Custom Gate Definition, Message Gate Definition, Custom Event Gate Definition</td>
</tr>
<tr>
<td>Support for soft coprocessors</td>
<td>Coprocessor Implementation Definition, Coprocessor Block Optimization Definition, Coprocessor Scheduler Definition</td>
</tr>
<tr>
<td>Support for accelerators</td>
<td>GP Implementation Definition, GP Block Optimization Definition</td>
</tr>
<tr>
<td>Support for GUIs</td>
<td>GUI Application Definition, Custom Representation Definition</td>
</tr>
</tbody>
</table>

8.6 DSPE Model Aspects

As introduced in Section 2.4, DSPE provides model Aspects. DSPE uses the following approach: at model level, the information of crosscutting features is stored in model Entities that extend the Aspect abstract model type (see Section 8.2). Model Entities that derive from Aspect are centralized places for storing the Aspect information, which is associated with the top-level Definitions that the Aspects impact. Then, when generation takes place, the parts of code concerning the top-level Definition and all other referenced Definitions are produced, and, contemporaneously, the Aspect’s related source code is distributed and integrated with the rest of the software architecture. As explained in Section 2.4, generators may perform rather complex transformations between abstract model information and source code. Therefore, in DSPE, Aspects are mostly used for application-wide features, for global optimizations, for customizations, for profiling, and for debugging.

For the purpose of developing generators and functionalities of the DSPE development envi...
ronment, DSPE provides special-purpose infrastructure for Aspects. For instance, support is available for retrieving all Aspects associated with a Definition, for finding all Aspects of a certain type present in a System and for aspect-oriented navigation of model elements, by traversing the model information depending on the Aspects values (see Section 9.4 for additional details).

Furthermore, to allow the specialization of Aspects in DSPE sub-models, the DSPE infrastructure for Aspects features functionality for substituting and extending Aspects. If a sub-model extends another sub-model, it is possible to

**Override Aspects** if the extending sub-model is not enabled, the Aspect of the parent sub-model is used. Otherwise, if the child model is enabled, the overriding Aspect completely substitutes the Aspect in the parent sub-model.

**Overload Aspects** if the extending sub-model is not enabled, only the information of the parent Aspect will be available. Instead, if both sub-models are used, the properties of both Aspects are merged and contemporaneously provided.

The DSPE model features the following Aspects:

- **Block Processing Aspect**: used to manage the support of block processing (see Section 3.19 for details). The Block Processing Aspect is associated with Structures, it is present in the ground sub-model, and it is overloaded in the XT sub-model to introduce the functionality for variable block processing. More information about the Block Processing Aspect is available in Section 8.11.

- **Operational Aspect**: used to simultaneously switch between the Simulation and the Production version of the Unit Behaviours associated with the Units in a Structure (see Section 8.7 for details about Simulation and Production). Furthermore, it provides support for bypassing Units from generation. The Operational Aspect is associated with Structures and is available in the base sub-model. More information about the Operational Aspect is available in Section 8.12.

- **Runtime Aspect**: features properties for configuring the execution infrastructure of the generated applications. For instance, it allows enabling the console/log information and execution profiles (see Section 3.25). It is associated with Runners, it is available in the base sub-model, and it is overloaded in the GUI sub-model. More information about the Runtime Aspect is available in Section 8.14.

- **Managers Aspect**: allows configuring the DSPE managers (see Section 3.23 for details), in particular the thread manager and the coprocessor manager. The Managers Aspect is associated with Runners, is available in the base sub-model, and is overloaded in the XT sub-model. More information about the Managers Aspect is available in Section 8.15.
8.7 Software Unit Definitions and Composite Definitions

According to Section 3.1, software components in DSPE are Units and Runners. As is the case for the generated source code, for the DSPE model there are two types of Unit Definitions: Software Unit Definitions and Composite Definitions. Similarly, some types of Runner Definitions are available (see Section 8.13 for more information). At model level, Units are specified as black-box components. By means of Gate Declarations, it is possible to specify the Unit's inputs and outputs, which are subsequently used when connecting Units in Structures.

To support the infrastructure described in Section 3.10, Gate Declarations may be either of type Parameter or Data, and to support events (see Section 5.3) Event Gate Declarations may be added. Furthermore, in Units, it is possible to specify the size of Group Gates (see Section 3.9 for more details).

Additionally, for each Unit, the DSPE model allows specifying Archetypes. An Archetype is used for extending and specializing a Unit in a way similar to how a child class extends and specializes a parent class in an object-oriented programming language. Multiple inheritance is allowed by referencing more than one Archetype at the same time.

The internal functionality of Units is specified by referencing Unit Behaviour Definitions for Software Units and Structure Definitions for Composites. Two Unit Behaviour Definitions are allowed for each Software Unit (see Figure 8.4). The first one is called Simulation and the second one Production. Instead, Composites allow a single Structure, which may be either a Configuration, a Scheduler or a Coprocessor Scheduler.

From the outside, Composite Definitions are identical to Software Unit Definitions. Internally,
8.7. Software Unit Definitions and Composite Definitions

A Composite Definition is instead designed by specifying the contained Structure Definition and by connecting the Gates of the Structure's Units with the Gates of the Composite, by means of Internal Links (see Figure 8.5). Then, the Composite Definition can be used in any other Structure Definition as if it were a Software Unit Definition. In addition, similarly to Structure Definitions, Composite Definitions allow adding and connecting constant values. For more information about Links and constant values, see Section 8.9.

In Extended Backus–Naur Form (EBNF), a Software Unit Definition can be formulated as

\[
\text{Software Unit Definition} = \text{name}, \\
\text{Parameter Gates}, \\
\text{Data Gates}, \\
\{\text{Simulation Unit Behaviour Dref.}\}, \\
\{\text{Production Unit Behaviour Dref.}\}, \\
\{\text{Archetype Unit Dref.}\};
\]  

(8.1)

where \text{name} is a string of characters (contained in a Field at the base model level) and \text{Definition References}, abbreviated with \text{Dref.}, are implemented with Relations.

In Formulation 8.1

\[
\text{Parameter Gates} = \{\text{Parameter Input Gate Declaration}\}, \\
\{\text{Parameter Output Gate Declaration}\};
\]  

(8.2)

\[
\text{Data Gates} = \{\text{Data Input Gate Declaration}\}, \\
\{\text{Data Output Gate Declaration}\};
\]
where each Gate Declaration can be formulated as

\[
Gate\ Declaration = \text{name, Gate Dref};
\]  

(8.3)

Instead, in EBNF, a Composite Definition can be formulated as

\[
Composite\ Definition = \text{name,}
\begin{align*}
& Parameter\ Gates, \\
& Data\ Gates, \\
& Structure\ Dref, \\
& Parameter\ Internal\ Links, \\
& Data\ Internal\ Links, \\
& Constant\ Values;
\end{align*}
\]  

(8.4)

where, in addition to the elements specified above,

\[
Parameter\ Internal\ Links = \{\text{Parameter\ Input\ Link\ Sub\ Entity}\},
\{\text{Parameter\ Output\ Link\ Sub\ Entity}\};
\]

(8.5)

\[
Data\ Internal\ Links = \{\text{Data\ Input\ Link\ Sub\ Entity}\},
\{\text{Data\ Output\ Link\ Sub\ Entity}\};
\]

and

\[
Constant\ Values = \{\text{Parameter\ Constant\ Value\ Sub\ Entity}\},
\{\text{Data\ Constant\ Value\ Sub\ Entity}\};
\]

(8.6)

All Internal Link Sub Entities of Formulation 8.5 can be expressed as

\[
\begin{align*}
Input\ Link\ Sub\ Entity &= \text{Source\ Gate\ dref,} \\
& \quad \text{Target\ Unit\ dref,} \\
& \quad \text{Target\ Gate\ dref.}
\end{align*}
\]

(8.7)

\[
\begin{align*}
Output\ Link\ Sub\ Entity &= \text{Source\ Unit\ dref,} \\
& \quad \text{Source\ Gate\ dref,} \\
& \quad \text{Target\ Gate\ dref.}
\end{align*}
\]

where Declaration References, abbreviated with dref., are implemented with Handles.
Constant Value Sub Entities are instead

\[
\text{Constant Value Sub Entity} = \text{value, Target Unit dref, Target Gate dref},
\]

where \text{value} is any value compatible with the type of the target Gate.

For more details about the information that may be specified with Software Unit Definitions and Composite Definitions see the UML diagram in Annex A.

Composite Definitions allow reuse of model parts. It is possible to use more instances of the same Composite Definition in one or more Structure Definitions, and it is possible to reuse the same Structure Definition inside more than one Composite Definition. Therefore, complex solutions with many levels of composition are possible. For the purpose of simplifying Composite management, DSPE provides model-based transformations that allow automatically creating and collapsing Composites (see Section 9.8 for more details).

From the abstract information specified with Software Unit Definitions and Composite Definitions, the DSPE generators produce source code that provides the functionalities described in Section 3.8, Section 3.13, and Section 3.21. In particular, generators create the source code of the C structs representing each specified Software Unit Definition or Composite Definition. For each Gate Declaration present at model level, generators also include references to memory areas, as part of the C structs. Furthermore, generators produce a specific function for each lifecycle phase of the Unit (see Section 3.15). If the specified Software Unit Definitions or Composite Definitions contain Event Gate Declarations, generators also create the infrastructure for receiving and sending events (see Section 5.5 and Section 5.12) and, if enabled, the infrastructure for managing events automatically (see Section 5.10). The same applies to the soft coprocessor infrastructures (see Chapter 6). Some parts of this infrastructure, such as support for special-purpose, task-ready events and support for pull and push execution pressure, are generated from Software Unit Definitions as part of Coprocessor Units (see Section 6.2).

On the other hand, the source code generated for the various types of Units works in close cooperation with the source code generated for the contained Unit Behaviours or Structures. Therefore, when generating Units, the information in associated Unit Behaviour Definitions and Structure Definitions may also be used.

### 8.8 Gate Definitions

Gate Definitions constitute the type system of DSPE. Frequently, new Gate types need to be specified and used in the application’s Units for fulfilling specific requirements of vertical application domains. In DSPE, Gates are a powerful means for extending the DSPE DSL (see Section 2.7 for an introduction about this topic). Furthermore, Gate Definitions are one of the
Chapter 8. DSPE Model

model parts allowing integration with C, C++, and assembly source code.

DSPE supports four categories of simple Gate Definitions (Standard Gate, String Gate, Custom Gate, and Pointer Gate) and five categories of Event Gate Definitions (Signal Gate, Event Gate, String Message Gate, Custom Event Gate, and Message Gate). Standard Gates and Event Gates may be of the following standard types: boolean, char, int, float, double, short, long, long double, and long long. The unsigned variants of char, short, int, long, and long long are also supported. String Gates and String Message Gates are used for sequences of characters. Custom Gates and Custom Event Gates are used for user defined data types. Pointer Gates and Message Gates support pointers.

Each Gate Definition provides properties and constraints for specializing the Gate's data type. Properties and constraints are particularly useful to avoid invalid and uninitialized values. For instance, it is possible to specify default values, minimum values, maximum values, and value intervals. Furthermore, Gates may be defined as arrays instead of individual values.

To allow connecting Gates of different types, but with compatible values, Gates Definitions allow specifying compatible Gates.

For more details about the information that may be specified with Gate Definitions see the UML diagram in Annex A.

From the abstract information provided with Gate Definitions, the DSPE generators produce source code for allocating, managing, and disposing the memory regions associated with Gates (see Section 3.9 for details). In addition, source code is produced for verifying that the value constraints are respected. For all types of Event Gates, the infrastructure for event pools is also generated (see Section 5.4). Then, the source code generated for Gate Definitions is used by all the DSPE software components, which contain the related Gate Declarations.

8.9 Configuration, Scheduler, and Coprocessor Scheduler Definitions

Configuration Definitions are the simplest type of Structure Definition featured by DSPE. Instead, Scheduler Definitions and Coprocessor Scheduler Definitions are evolved versions that support event-driven scheduling and parallel processing.

All types of Structure Definitions are combinations of interconnected Unit Declarations (see Figure 8.6). Each Unit may exchange information with any number of other Units by means of Links. Links are communication channels set up between compatible Gate Declarations. Links may be added either between Data Output Gates and Data Input Gates, or between Parameter Output Gates and Parameter Input Gates. In all types of Structure Definitions, it is possible to build connection cycles for feedback loops, which are, for instance, frequently present in audio processing applications. Structure Definitions also allow adding and connecting constant values to the Gate Declarations of Unit Declarations. These constant values are set by the source code at initialization time.
In the generated source code (see Section 3.12, Section 5.8 and Section 6.7), Configurations differ from Schedulers mainly in terms of the approach used for executing the contained software components. For the process lifecycle phase Schedulers use an event-driven approach. While Configurations use a fixed order of execution for all lifecycle phases, Schedulers exceptionally executes the process phase according to the flow of events. Therefore, in the DSPE model, it is possible to specify a default execution sequence for each Structure Definition, which is used by Configurations for all lifecycle phases and by Schedulers for all phases except the process one. In addition, Sequence Definitions may be used for specifying alternative execution orders (see the next section for some details).

Compared with Configuration Definitions, Scheduler Definitions have additional support for specifying special-purpose initialization events (see Section 5.8) and policies to use for idle time and suspension (see Section 5.14). Coprocessor Scheduler Definitions are special-purpose versions of Scheduler Definitions. Coprocessor Scheduler Definitions have to be used if the soft coprocessor infrastructure or the accelerator infrastructure are required (see Chapter 6 and Chapter 7 for more details). At model level, they additionally provide settings for managing special-purpose task-ready events (see Section 6.4).
Chapter 8. DSPE Model

In EBNF the principal part of a Structure Definition can be formulated as:

\[
\text{Structure Definition} = \text{name},
\{\text{Unit Declaration}\},
\text{Links},
\text{Constants},
\text{Sequences};
\] (8.9)

where, in addition to the elements specified in Section 8.7:

\[
\text{Unit Declaration} = \text{name}, (\text{Unit Dref.} \mid \text{Composite Dref.});
\] (8.10)

\[
\text{Links} = \{\text{Parameter Link}\}, \{\text{Data Link}\};
\] (8.11)

\[
\text{Sequences} = \{\text{Start-up Sequence Dref.}\},
\{\text{Pre-process Sequence Dref.}\},
\{\text{Process Sequence Dref.}\},
\{\text{Post-process Sequence Dref.}\},
\{\text{Shutdown Sequence Dref.}\};
\] (8.12)

Each Link Sub Entity of Formulation 8.8 can be expressed as

\[
\text{Link Sub Entity} = \text{Source Unit dref.},
\text{Source Gate dref.},
\text{Target Unit dref.},
\text{Target Gate dref.};
\] (8.13)

For more details about the information that may be specified with the different types of Structure Definitions see the UML diagram in Annex A.

The source code generated from the various types of Structure Definitions includes all C structs that represent the Structure Definitions in the source code (see Section 3.12 for an example) and all functions for managing their lifecycle phases (see Section 3.15). In addition, for Scheduler Definitions, the event-driven support is generated, which is the most substantial part of the overall event-driven infrastructure generated by DSPE (See Chapter 5, in particular Section 5.8). Concerning Coprocessor Schedulers, from the information provided at model level, the DSPE generators also produce the infrastructure for managing tasks and the special-purpose, task-ready events (see Section 6.4 and Section 6.7 for more information).
8.10 Sequence Definitions

In all situations, where an out-of-the-ordinary order has to be specified for one of the execution lifecycle phases of Structures, it is possible to specify a Sequence Definition referenced by the requiring Structure. Sequence Definitions provide a solution, for instance, if a particular Unit has to be initialized for first at start-up, but needs to be finalized as the last during shutdown. With Sequence Definitions, it is possible to specify alternative execution orders for all lifecycle phases of Structures, apart from the process phase of Schedulers and Coprocessor Schedulers, which execute with event-driven approach.

The DSPE generators use the abstract information featured by Sequence Definitions when generating Configurations, Schedulers, and Coprocessor Schedulers. No file of source code is directly generated from Sequence Definitions.

8.11 Block Processing Aspect

The Block Processing Aspect is the DSPE model Aspect used for managing the block processing support of DSPE (see Section 3.19 for more information). For each type of Structure, it is possible to specify the behaviour of the block processing functionality by enabling and disabling blocks and by specifying the block size. In some situations block processing is useless or may excessively waste memory; therefore, it is possible to disable it. In addition, the Block Processing Aspect allows configuring variable block processing, which may be useful if the data streams to be processed have an irregular rate.

The DSPE generators use the abstract information specified in the Block Processing Aspect for producing specific source code parts that are distributed in Structures, Composites, and Runners. Furthermore, the information is used for correctly including the generated code of Units. As introduced in Section 3.5, more than one version of the Units may be generated, depending on the overall utilization of block processing.

8.12 Operational Aspect

The Operational Aspect allows switching all Unit Behaviours of Software Units, from Production to Simulation and vice-versa (see Section 8.7 for more information). Furthermore, if required, the Operational Aspect allow bypassing the generation of the Units contained in Structures, by means of two groups called “Simulation Omissions” and “Production Omissions”.

The Operational Aspect influences the generated source code in a distributed way. In particular, similarly to the Block Processing Aspect, its information is used for including the correct version of generated source code of the Units.
8.13 Runner Definitions

At the model level, Runner Definitions include Structure Definitions in the same way as done by Composite Definitions (see Section 8.7). However, Runner Definitions are used to specify executables. As a consequence, unlike Composite Definitions, a Runner Definition cannot be used inside other Structure Definitions. Furthermore, Runner Definitions can only have parameter Gates. These Gates are used for specifying how the processing engines communicate with the generated user interfaces. As a consequence, the abstract information provided by these Gates is also used for producing the critical section of applications (see Section 3.22).

Two main types of Runner Definitions are available in DSPE: Application Definition and GUI Application Definition. Application Definitions have to be used for generating Command-Line Runners, Shell Runners, Custom Runners, and Empty Runners. GUI Application Definitions are instead special-purpose Definitions for specifying GUI Runners. GUI Application Definitions additionally allow specifying the type of graphical components, such as buttons, sliders, or combo-boxes to use for the Gate Declarations. Section 3.14 provides more information about the source code, which is possible to generate for Runners.

By the help of the elements specified in Section 8.7, in EBNF, the principal part of a Runner Definition can be formulated as:

\[
\text{Runner Definition} = \text{name, Parameter Gates, Structure Dref., Parameter Internal Links, Constant Values; (8.14)}
\]

The abstract information provided by Runners is used by the DSPE generators for producing many parts of the application source code, including the user interfaces (see Section 3.14) and all managers (see Section 3.23). However, the abstract information on Runners is also used for creating the executables, by means of the integration support provided by the DSPE development environment (see Section 9.9 for more information).

8.14 Runtime Aspect

The Runtime Aspect features a multitude of settings useful for configuring the behaviour of Runners and for activating support for debugging and optimization. For instance, by means of the Runtime Aspect, it is possible to set the critical section update rates (see Section 3.22) and to enable functionalities such as the console/log information and execution profiles (see Section 3.25 for more information).
8.15 Managers Aspect

In a way similar to the Runtime Aspect, the Managers Aspect collects settings useful for configuring the behaviour of managers (see Section 3.23). Most of the provided settings concern the threading support (see Section 3.24 and Section 6.19) and the soft coprocessor support (see Chapter 6). For example, the Managers Aspect allows selecting the threading system to use, enabling thread affinities, and configuring the policy for assigning the affinities to the threads (see Section 6.19). Similarly, it allows configuring many properties of task queues and pools, such as minimal and maximal sizes (see Section 6.4 and Section 6.5) and the behaviour of the scheduler contribution (see Section 6.15).

8.16 Unit Behaviour Definitions

As introduced in Section 2.7, the development of new domain-specific software components—for instance, for signal compression, cryptography or neural networks—allows extending the DSPE DSL to vertical application domains. Therefore, special-purpose support is required, in particular, for integrating preexisting functionalities in external libraries and frameworks. To solve this problem, DSPE provides source code templates featuring the default behaviour of software components (see Section 3.16 for introductory details). As a consequence, the DSPE model provides various types of Unit Behaviour Definitions, which allow configuring and generating these templates. After generation, templates may be used for integrating and developing algorithms in C, C++, or assembly. Together with Gates, Unit Behaviour Definitions are part of the functionalities of the DSPE model that allow the synergy of general-purpose languages and the DSPE DSL.

Unit Behaviour Definitions provide many configuration features. In general, they allow specifying the internal variables and the different lifecycle phases of Software Units (see Section 3.15 for details). Furthermore, these Definitions allow specifying Requirements (see Section 8.18) and enabling/disabling access to the manager functionalities (see Section 3.23). In addition

State Unit Behaviour Definitions allow specifying the special-purpose internal state required for developing finite-states machines and the list of possible values of this state (see Section 5.9 for more details). Furthermore, State Unit Behaviour Definitions feature functionalities for configuring the automatic support for events (see Section 5.10). These functionalities include flags for progressively enabling and disabling the automatic features and special-purpose groups for bypassing Event Gates from being automatically managed. In addition, the functionalities for configuring pull and push management of the application execution pressure are also provided (see Section 6.16). These functionalities are mostly used in Coprocessor Unit Behaviour Definitions, which extend State Unit Behaviour Definitions, but also allow using State Unit Behaviours in combination with Coprocessor Unit Behaviours and GP Unit Behaviours in parallel stream-processing applications.
Chapter 8. DSPE Model

Coprocessor Unit Behaviour Definitions extend all the functionalities provided by State Unit Behaviour Definitions. In addition, they allow configuring the part of the soft coprocessor infrastructure that is integrated in Coprocessor Units and Coprocessor Unit Behaviours (see Section 6.2 and Section 6.8 for details). In particular, Coprocessor Unit Behaviour Definitions allow specifying the sizes of the queues between Coprocessor Unit Behaviours and soft co-processors and enabling/disabling the functionalities for automatically managing tasks (see Section 6.10). Furthermore, Coprocessor Unit Behaviour Definitions allow configuring block decompositions and aggregations (see Section 4.2).

GP Unit Behaviour Definitions extend all the functionalities provided by Coprocessor Unit Behaviour Definitions and, in addition, allow integrating CUDA kernels, which the generated source code executes on GPUs (see Chapter 7 for more details). All properties for configuring the kernel execution and all additional lifecycle phases required—for instance, for allocating, managing and disposing the device memory—are available as part of the GP Unit Behaviour Definitions. GPU Unit Behaviour Definitions also provide support for specifying alternative algorithm implementations to use when performing heterogeneous work stealing.

Wrap Unit Behaviour Definitions extend the functionalities provided by Coprocessor Unit Behaviour Definitions and, in addition, allow specifying the information for programmatically managing Composites (see Section 3.18 for more details).

In agreement with the functionalities provided by the source code (see Section 3.16), in the DSPE model, there are two forms of Unit Behaviour Definitions for each of the types specified above. Therefore, there are Implementation Definitions, Block Optimization Definitions, State Implementation Definitions, State Block Optimization Definitions, Coprocessor Implementation Definitions, Coprocessor Block Optimization Definitions, GP Implementation Definitions, GP Block Optimization Definitions, Wrap Implementation Definitions, Wrap Block Optimization Definitions. As explained in Section 3.19 Implementations feature additional functions for their lifecycle phases. In addition, at model level it is possible to associate a Block Optimization Definition to every Implementation Definition. If this association is present, the DSPE source code generators are able to use the correct version of Unit Behaviour, depending on the value specified for the Block Processing Aspect. In other words, if block processing is disabled, the source code generated from the Implementation Definition is used. Instead, if block processing is disabled, the source code generated from the Block Optimization Definition is executed. This solution provides an important complement to the infrastructure described in Section 3.19 because allows avoiding the use of suboptimal types of Unit Behaviours.

For more details about the information that may be specified with Unit Behaviour Definitions see the UML diagram in Annex A.
8.17 Runner Behaviour Definitions

Similarly to Unit Behaviour Definitions, DSPE also allows specifying Runner Behaviour Definitions. At the source code level, the infrastructure for Runner Behaviours is deeply integrated with the rest of the generated source code (See Section 3.17). Runner Behaviour Definitions are instead straightforward to use and mostly provide functionalities for specifying the application's global variables and its lifecycle phases. Furthermore, like Unit Behaviour Definitions, Runner Behaviour Definitions allow specifying Requirements (see the next section) and enabling/disabling access to the manager functionalities (see Section 3.23).

8.18 Requirement Definitions

A Requirement Definition is a special-purpose type of Definition, which allows integrating external parts of source code within the DSPE model. In particular, Requirement Definitions feature paths to source code files and libraries used by the compiler for properly building the application executable. Mostly, requirements are used in Gate Definitions and Unit Behaviour Definitions, when integrating source code written in C, C++, or assembly in the DSPE DSL.
DSPE Development Environment

DSPE is a development environment for prototyping and customizing parallel real-time stream-processing applications, which is composed of plug-ins for the Eclipse development platform [42]. As introduced in Chapter 8, its domain-specific model is based on EMF [44, 126], while, its editors and wizards are based on frameworks provided by Eclipse, such as the Graphical Editing Framework (GEF), the Graphical Modelling Framework (GMF), the JFace UI toolkit, and the Eclipse infrastructure for forms. On the other hand, the DSPE generators are developed with Java Emitter Templates (JET) and XPand template languages, which are part of the Eclipse modelling tools [44]. In addition, DSPE integrates with CDT [43] for direct C/C++ development and for application compilation and debugging. As part of its functionalities, DSPE also supports model validation, model-level refactoring, and model transformations. These functionalities stand on top of the DSPE model and are provided in the development environment, by means of contextual actions and eclipse wizards.

This chapter provides an overview of the DSPE development environment and briefly discusses the infrastructures and frameworks used for its development. All the functionalities provided by DSPE have the goal of facilitating and accelerating the development of parallel stream-processing applications that are based on the infrastructures for event-driven scheduling and dynamic load balancing described in Chapter 5, Chapter 6, and Chapter 7.

9.1 Domain-Specific Development Techniques and DSPE

As introduced in Chapter 2, DSPE allows prototyping and customizing of stream-processing applications by means of model-driven code generation. DSPE uses model-driven development for collecting and managing the abstract information of software components and of all features of the generated software architecture. Therefore, in DSPE, the DSPE model is used as a centralized repository by all other parts of the development environment. On the other hand, generators utilize generative programming techniques for transforming the abstract information provided by the model into executable source code. Source code is produced for components, for the software architectures of the applications, for textual and graphical user
interfaces, and for all other functionalities needed by the applications. Component-based, pattern-based, and aspect-oriented techniques are used both at the model level and in the generated code, for combining model elements into application specifications and source code parts into generated applications.

The graphical user interface of DSPE also takes advantage of model-driven development. It is composed of editors and wizards that act on the model directly and allow for navigating and modifying the contained abstract information. Furthermore, the DSPE user interface allows for performing source code generation of the designed applications. The synergic combination of the DSPE model, its user interface, its generators, and the assisting functionalities transforms DSPE into a specialized visual environment, where stream-processing applications may be designed by focusing on their fundamental domain-specific features.

9.2 Editors, Wizards, and Other Parts of the DSPE User Interface

For each Definition of the DSPE model (see Chapter 8), DSPE features a dedicated editor as part of its graphical user interface. There are two types of editors in DSPE: visual editors and form editors. Visual editors have been developed with the GEF and GMF frameworks. These frameworks are provided by Eclipse and allow for the creation of editors with visual elements that may be moved, combined, and connected on a 2D surface. Furthermore, support for undo/redo and for contextual actions on the visualized elements is provided. On the other hand, form editors have been developed with the Eclipse infrastructure for forms and with JFace. As a consequence, the DSPE form editors mostly allow editing the information by means of check boxes, combo-boxes, selection boxes, buttons, and textual areas.

In DSPE, visual editors are provided for the model Definitions that have a visual representation in the DSPE DSL. These are Software Units, Composites, all types of Structures, and all types of Runners. Form editors are, instead, provided for all parts of the DSL that have textual representations, such as all types of Gates and Unit Behaviours. In addition, all DSPE editors provide an outline and a properties view, which alternatively allow editing the information in the DSPE model in tree structures.

The DSPE wizards have also been developed with JFace. All wizards look similar to form editors but, unlike form editors, they are not directly associated with a particular Definition. The main wizards are used for creating new Systems and new Definitions. Furthermore, dedicated wizards are featured for the most advanced model transformations, in particular for the profile-guided model transformations.

The remaining parts of the DSPE user interface are represented by a navigator view specific for Systems and by buttons and contextual actions that allow performing generation, model validation, various types of model refactoring, and all model transformations. In addition, an Aspects overview is provided for navigating the Definitions contained in a System in an aspect-dependent way.
9.3 Infrastructure Used for Developing DSPE Generators

In DSPE, a dedicated generation engine executes generators, which query DSPE model Definitions and produce source code. The generation engine is also responsible for creating file system elements such as projects, folders, and files. The DSPE generators are developed with three solutions for writing model-to-text transformations:

- **A framework of Java classes** has been explicitly developed for DSPE, with the purpose of modularizing the DSPE generators at a coarse-grain level. It provides infrastructure for combining and reusing generator parts, and it features mechanisms for conditional and repetitive generation. By taking advantage of inheritance and composition, these java classes allow specifying abstract elements of generators, which may be subsequently specialized to fit specific needs.

- **Java Emitter Templates (JET)** is a utility part of the Eclipse modelling tools, used in DSPE for implementing transformation templates at a fine-grain level. They allow editing static and dynamic source code parts used at generation time for producing the source code. JET has been integrated with the DSPE generation engine and with the DSPE generation framework of Java classes described above.

- **XPand** is another solution part of the Eclipse modelling tools, utilized by DSPE for developing model-to-text transformation templates. In a way similar to JET, it has been integrated with the DSPE generation engine and framework. It has been used for conceiving various parts of the DSPE generators, in particular GUI generators.

9.4 Specific Support for Querying and Traversing the DSPE Model

To facilitate the development of the DSPE generators and all assisting functionalities, such as model validation, refactoring, and transformations, dedicated infrastructure has been developed, which allows

1. **Querying the DSPE Definitions** in a specific way. For example, there are functions for retrieving all Gates of a particular type, such as all Event Gates, which may be contained in a Software Unit, a Composite, or a Structure. As another example, there are functions for collecting either all Parameter Links or all Data Links used in Structures, Composites, and Runners. All these functions are contained in Java classes accessible from generators and from all other parts of the DSPE development environment.

2. **Traversing DSPE Definitions** by using the visitor pattern. Abstract visitors are provided, which are specific for DSPE Definitions and which allow, for instance, navigating a complete application description from its Runner Definition to its Gate Definitions. Visitors may be used for collecting information or for modifying the Definition contents while traversing the application descriptions. The provided infrastructure allows navigating the application specifications in an aspect-dependent way.
9.5 Implementation of the DSPE Generators

The DSPE generators have been developed with the infrastructure and the specific support described in the previous sections. Each generator is associated either with a specific type of Definition or with a System. More than one generator may exist for each type of Definition or System. For each generator, it is possible to specify a generation condition, which allows specializing generators, for instance, to a specific type of Definition, to Definitions with particular properties, or to Systems containing certain types of Definitions.

Generators may be simple and composed only of a few Java classes and JET or XPand templates. However, most of the DSPE generators are complex and aggregate many sub-parts. A specific Java class, required by each generator, is used for registering the generator in the DSPE generation engine. This class is also responsible for testing the generation condition and, if required, for consequently executing the code of the other parts of the generator. During generation, the generation engine executes the code of this class for all registered generators on all selected Definitions of the supported Definition type.

Internally, the DSPE generators have a tree structure (see Figure 9.1 for a simple example). Classes at the roots of the tree are mostly responsible for querying the Definition being generated and, if required, all the Definitions it references. These classes collect information by directly accessing the Definition information or by using the DSPE-specific query function and the DSPE visitors described in the previous section. At the leaves of the generator trees, there are instead model-to-text templates that are executed iteratively and conditionally to produce the required source code. Each template combines static parts of source code with dynamic parts derived from the information collected from the Definitions. At the end of the generation process, all parts of the generated source code are assembled, and the result is written to file.

Figure 9.1: Example of the internal structure of a generator.
9.6 DSPE Model Validation

Syntactic and semantic validation of the abstract information contained in the DSPE Definitions is one of the most effective functionalities that assist and accelerate the design and implementation at model level. DSPE provides specific validation rules, which range from basic consistency checks of connections between components to complex verification of invalid recursive structures of the specified software architecture. Semantic validation accelerates development because it reduces the needs of application generation and tests when verifying the consistency of the application design.

In DSPE, all validation rules have been implemented with the EMF infrastructure for model validation. This infrastructure provides base abstract classes that may be extended and used for verifying the consistency of EMF model elements. Furthermore, the DSPE validation rules extensively uses the DSPE specific query function and the DSPE visitors introduced in Section 9.4.

9.7 DSPE Model Refactoring

Refactoring is a development technique that is normally performed at source code level with the goal of cleaning up code and improving its structure [49]. In modern software engineering, refactoring is an essential development task executed all along the development period. As an alternative to refactoring functionalities at source-code level, DSPE features refactoring at model level with the purpose of facilitating revision and evolution of the application abstract design. Frequent development and refactoring iterations performed at model level, speed up the application development process. Model refactoring is easier than source-code refactoring because, at model level, the amount of information that has to be managed is smaller than at source-code level.

Normally, refactoring implies that more than one Definition is simultaneously and consistently modified to avoid corrupting the application structure and behaviour. For instance, if the name of a component is modified during refactoring, then all references to the component also have to be updated in other Definitions. In DSPE, refactoring operations allows renaming and moving model elements between systems without loss of consistency. Furthermore, DSPE provides morph operations that allow converting the type of DSPE Definitions into other types that are compatible. For example, in DSPE, it is possible to convert a Configuration Definition into a Scheduler Definition (see Section 8.9 for more information).

One of the most useful morph operations provided by DSPE allows converting a Software Unit into a Composite and vice-versa. This functionality allows exploiting top-down development approaches side by side with bottom-up development approaches. DSPE users may initially add Software Units to Structures without promptly specifying their internal functionality. If required, the Software Unit may be converted into a Composite for which the internal Structure may be later specified.
Chapter 9. DSPE Development Environment

9.8 DSPE Model Transformations

The refactoring functionalities introduced in the previous section are provided by DSPE for facilitating the restructuring and simplification of the application designs. Instead, to facilitate applications enhancement, DSPE features model-to-model transformations. Unlike refactoring, model transformation also allows significant automatic evolution of the application specifications. The following categories of transformations are provided:

Deriving transformations allow creating new Definitions that are automatically populated with information derived from already-existing Definitions. For instance, a transformation is provided that facilitates the creation of a Runner Definition or a Composite Definition from a Configuration Definition, Scheduler Definition, or Coprocessor Scheduler Definition. During the transformation, Gate Declarations are automatically created and added to the Application Definition or Composite Definition for all Gate Declarations that are not linked in the source Definition.

Aggregating and splitting transformations allow combining and separating software components. For example, a transformation is provided that allows extracting a Composite from a Structure by selecting some of the contained Units. Similarly, a transformation allows collapsing a Composite in all the Structures in which it is contained.

Enhancing transformations allow creating groups of Definitions from other groups of Definitions to enhance the behaviour of entire parts of the applications. For instance, an eventizing transformation may be used for creating an application part based on events and made of Scheduler Definition, Units Definitions, and Gate Definitions, from an application part based on sequential execution and having a Configuration Definition at its root. Before transformation takes place, a special-purpose wizard allows selecting the Gate Declarations of the Configuration Definition and of its Units, for which the type has to be converted into Event Gates. The transformation autonomously creates all required new model elements.

Profile-guided transformations are special-purpose versions of enhancing transformations, which perform the transformations by also using the information contained in execution profiles of the application, as a complement to the information provided in the model (see Section 3.25 for information about the DSPE profiling infrastructure). The profile-guided transformations featured by DSPE facilitate the optimization and conversions of serial application versions into parallel processing versions.

9.9 Round-Trip Development and CDT Integration

As introduced in Section 2.7, Section 3.16, and Section 8.16, DSPE allows the integration of C, C++, and assembly source code directly within the DSPE model. This type of support may be used for optimizing and fine-tuning the designed stream-processing applications, but also for extending the DSPE functionalities to vertical application domains. For this purpose, DSPE
features functionalities for extracting—from the generated source code—specific parts of code that have to be encapsulated in the model. These functionalities are particularly useful for extracting the implementation of all functions for the lifecycle phases of Unit Behaviours. These functionalities favour round-trip development approaches. It is possible to develop the application specification at model level, generate the code, modify the relevant parts of the generated code, extract the modified source code, and restart from the model level.

As a complement to these functionalities, DSPE features integration with the CDT Eclipse plug-in [43] that allows editing, compiling, and debugging programs written in C, C++, and assembly. Consequently, it is possible to iteratively design, generate, compile, and execute the applications in a single development environment.

**9.10 Audio-Specific Software Components and Generator**

One of the most important vertical extensions of DSPE has been provided with Audio n-Genie in the domain of audio signal processing. This extension features domain-specific Gates and Units for audio processing applications, but it also features a dedicated source code generator and a small model enhancement. The main features of the Audio n-Genie extension are the following:

- Software components that utilize the *libsndfile* library [92]. These components allow reading and writing information from and to audio files.
- Software components that utilize the *PortAudio* library [113, 6]. These components allow reproducing and recording audio with sound cards.
- Software components that utilize the *Csound* library [30]. These components provide filters for audio processing and oscillators for audio signal synthesis.
- A dedicated source code generator for the *VST technology* by Steinberg. VST features a development kit for audio plug-ins that may be used in many digital audio workstations. This generator uses the abstract information provided by the DSPE model and, in addition, information specified with a dedicated model extension specific for VST plug-ins.

More information about Audio n-Genie is available on related publications [87, 88, 90] and on the DSPE web site [41].
9.11 Advantages of the DSPE Development Environment

All the features, such as editors, wizards, refactoring, and transformations, which are provided by the DSPE development environment and have been presented in this chapter, facilitate the design and the implementation of parallel stream-processing applications. Compared with traditional application development, the prototyping, refinement, and customization approach supported by DSPE introduces the following advantages:

1. Code reuse of components is favoured by their standardized communication interface and their black-box nature. Components may be implemented from scratch or may integrate pre-existing functionality from libraries and frameworks. This approach allows the extension of DSPE to vertical application domains.

2. In DSPE, design phases follow an agile approach by taking advantage of the flexibility and essentiality of the model and its validation rules. Compared with development phases at source code level, design phases at model level are shorter, and development iterations may be frequent. Evolutionary development processes are facilitated.

3. Growth and review of the design may be performed directly at model level where dedicated refactoring and automatic model-to-model transformations may be utilized. The generator will then automatically reflect all model changes in the source code.

4. Source code may contain parts for which the process of writing and maintaining by hand may be difficult or error prone. From the abstract information provided with the DSPE Definitions and Aspects, the DSPE generators are able to perform relatively complex transformations. This approach reduces the presence of defects and favours application performance and robustness.

5. Generated code, compared with handwritten code, is clean and contains only the needed functionalities. By disabling unnecessary features at model level, it is possible to obtain compact applications, both in terms of provided features and executable size.

6. Model-driven generative programming significantly reduces the available development variability and limits the producible application family to a known and tested set. The family of applications obtainable with generators is smaller than the family of applications programmable with a general-purpose programming language. Many applications of this second family may exhibit structural problems, poor performance, and bugs. The DSPE DSL and its generators help to filter them out.
A UML Diagram of the DSPE Model
<table>
<thead>
<tr>
<th>AdjustableBehaviourSupport [M] (XT)</th>
</tr>
</thead>
<tbody>
<tr>
<td>BehaviourType [f: UnitBehaviourType]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>WrapUnitBehaviourSupport [M] (XT)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Owner [f: OwnerType]</td>
</tr>
<tr>
<td>HasConsumeCompositeEvent [f: boolean]</td>
</tr>
<tr>
<td>ConsumeCompositeEvent [f: String]</td>
</tr>
<tr>
<td>WrappedComposite [r: Complex]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>GPImplementationSupport [M] (GP)</th>
</tr>
</thead>
<tbody>
<tr>
<td>KernelExecutionType [f: KernelExecutionType]</td>
</tr>
<tr>
<td>HasPreCopyToDevice [f: boolean]</td>
</tr>
<tr>
<td>PreCopyToDevice [f: String]</td>
</tr>
<tr>
<td>HasPostCopyToDevice [f: boolean]</td>
</tr>
<tr>
<td>PostCopyToDevice [f: String]</td>
</tr>
<tr>
<td>HasPreCopyFromDevice [f: boolean]</td>
</tr>
<tr>
<td>PreCopyFromDevice [f: String]</td>
</tr>
<tr>
<td>HasPostCopyFromDevice [f: boolean]</td>
</tr>
<tr>
<td>PostCopyFromDevice [f: String]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CoprocImplementationSupport [M] (XT)</th>
</tr>
</thead>
<tbody>
<tr>
<td>HasPreArmTask [f: boolean]</td>
</tr>
<tr>
<td>PreArmTask [f: String]</td>
</tr>
<tr>
<td>HasPostArmTask [f: boolean]</td>
</tr>
<tr>
<td>PostArmTask [f: String]</td>
</tr>
<tr>
<td>HasPreDismissTask [f: boolean]</td>
</tr>
<tr>
<td>PreDismissTask [f: String]</td>
</tr>
<tr>
<td>HasPostDismissTask [f: boolean]</td>
</tr>
<tr>
<td>PostDismissTask [f: String]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>StateUnitBehaviourSupport [M] (XT)</th>
</tr>
</thead>
<tbody>
<tr>
<td>EntryState [f: String]</td>
</tr>
<tr>
<td>AcceptInitEvent [f: boolean]</td>
</tr>
<tr>
<td>EventSupportType [f: EventSupportType]</td>
</tr>
<tr>
<td>HasAutomaticTransit [f: boolean]</td>
</tr>
<tr>
<td>HasAutomaticExecute [f: boolean]</td>
</tr>
<tr>
<td>HasAutomaticLoad [f: boolean]</td>
</tr>
<tr>
<td>HasAutomaticSend [f: boolean]</td>
</tr>
<tr>
<td>InputNextSupportType [f: NextSupportType]</td>
</tr>
<tr>
<td>OutputNextSupportType [f: NextSupportType]</td>
</tr>
<tr>
<td>ConsiderInitEventAsNext [f: boolean]</td>
</tr>
<tr>
<td>DisablePendingEventsSupport [f: boolean]</td>
</tr>
<tr>
<td>PendingEventsExpulsionPolicy [f: PendingEventsExpulsionPolicy]</td>
</tr>
<tr>
<td>MaxNumPendingEvents [f: int]</td>
</tr>
<tr>
<td>HasPendingEventsProportionalDelivery [f: boolean]</td>
</tr>
<tr>
<td>PendingEventsDeliveryRatio [f: int]</td>
</tr>
<tr>
<td>HasCompulsoryProcess [f: boolean]</td>
</tr>
<tr>
<td>PreCompulsoryProcess [f: String]</td>
</tr>
<tr>
<td>HasPostCompulsoryProcess [f: boolean]</td>
</tr>
<tr>
<td>PostCompulsoryProcess [f: String]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CoprocUnitBehaviourSupport [M] (XT)</th>
</tr>
</thead>
<tbody>
<tr>
<td>UnitBehaviourState [SG: AdditionalStateVariable]</td>
</tr>
<tr>
<td>TaskQueueType [f: TaskQueueType]</td>
</tr>
<tr>
<td>FavourCoprocEvents [f: boolean]</td>
</tr>
<tr>
<td>IgnorePreferenceQueueScheduling [f: boolean]</td>
</tr>
<tr>
<td>HasTaskBuffer [f: boolean]</td>
</tr>
<tr>
<td>HasAutomaticTaskManagement [f: boolean]</td>
</tr>
<tr>
<td>HasInternalSplitFactor [f: int]</td>
</tr>
<tr>
<td>InternalSplitFactor [f: int]</td>
</tr>
<tr>
<td>HasAutomaticTransfer [f: boolean]</td>
</tr>
<tr>
<td>EnableTaskBusyLimit [f: boolean]</td>
</tr>
<tr>
<td>TaskBusyMax [f: int]</td>
</tr>
<tr>
<td>OverrideCoprocQueueMax [f: boolean]</td>
</tr>
<tr>
<td>CoprocQueueMax [f: int]</td>
</tr>
<tr>
<td>CoprocQueueMinDependentNext [f: boolean]</td>
</tr>
<tr>
<td>OverrideCoprocQueueMin [f: boolean]</td>
</tr>
<tr>
<td>CoprocQueueMin [f: int]</td>
</tr>
<tr>
<td>CoprocQueueRestrainPolicy [f: CoprocQueueRestrainPolicy]</td>
</tr>
<tr>
<td>OverrideMultiTaskConsumptionPolicy [f: boolean]</td>
</tr>
<tr>
<td>MultiTaskConsumptionPolicy [f: TaskConsumptionPolicy]</td>
</tr>
<tr>
<td>TaskToConsumeratio [f: int]</td>
</tr>
<tr>
<td>NumTaskToConsume [f: int]</td>
</tr>
<tr>
<td>HasArmTask [f: boolean]</td>
</tr>
<tr>
<td>ArmTask [f: String]</td>
</tr>
<tr>
<td>HasDismissTask [f: boolean]</td>
</tr>
<tr>
<td>DismissTask [f: String]</td>
</tr>
</tbody>
</table>
Bibliography


Bibliography


