Abstract—The OSSS methodology defines a seamless design flow for embedded HW/SW systems. It enables the effective use of high-level SystemC™ and C++ features like classes (object-oriented design paradigm), templates and method based communication for the description of SW and HW. Furthermore, it supports the OSCI SystemC Synthesis Subset for low-level HW description and HW IP integration. With Fossy we provide a tool for the automatic transformation of a system description in OSSS to an implementation. In this paper we present a top-down design flow using the OSSS methodology for the implementation of an adaptive video filter. The last step of the proposed design flow has been performed automatically by Fossy. We have targeted a Xilinx FPGA to proof the usability of a physical implementation for future SoC designs.

I. INTRODUCTION

Today, highly integrated embedded systems have a wide range of usage in our daily life, such as telecommunications and automotive. These modern embedded systems consist of several IP blocks plus a few custom components and often use pre-defined technology platforms to implement them. Existing and upcoming technologies offer ever more possibilities to cope with the increasing application requirements. Exploiting these advanced platform capabilities and preserving as much of the platform independent application description, raises the demand for flexible design methodologies. The OSSS (Oldenburg System Synthesis Subset) methodology [1] aims to help the designer in separating pure application specific parts from implementation platform dependent information, thus enabling flexible mapping and refinement to different target platforms.

In our definition, embedded systems consist of an arbitrary number of: Software components to be executed on software processors, which mostly perform control and supervision tasks, Hardware components, that may perform computationally intensive or time critical tasks, IP (Intellectual Property) components, either 3rd party components or hardware and software components from previous projects, and Interface components that perform the communication between Hardware and Software.

It has been observed that the integration of these different pre-designed components into an efficiently working system is much more difficult than the design of a single component. This observation has massively influenced the principle of platform-based design [2].

In practice, most embedded systems are implemented on so-called technology platforms. These platforms provide a basic configuration consisting out of processors, memories, special hardware resources (or accelerators), communication peripherals, and communication resources (like busses, special cross-bar switches or Network-on-Chips) to connect them all together. The application is implemented on top of this platform configuration.

From a technical point of view the design of embedded HW/SW systems involves all steps from the initial specification of the application - what is the system supposed to do - to the efficient implementation on a technology platform - how is it implemented. The challenge consists in finding methods for the description and transformation/refinement of the initial specification to an efficient implementation.

From the industrial point of view the requirements for the design of embedded systems are very tight. Targeted development costs and time-to-market needs to be matched, but at the same time functional and non-functional product requirements need to be fulfilled. Consequences for the design process are: Modelling on the highest possible level of abstraction (mastering of complexity), most accurate analysis of system characteristics (prevention of costly re-design cycles), high degree of automation (raising efficiency and prevention of errors), and re-use of pre-existing components. These requirements have been considered during the development of the OSSS methodology. It enables separation of application and implementation platform, seamless refinement, executable models, IP integration and automatic synthesis.

The rest of this paper is organised as follows. Section II starts with a short discussion of related work. This is followed by the description of the OSSS methodology, including the proposed design and synthesis flow. The subsequent section illustrates the path from application to implementation, by first giving a quick overview of model composition and mapping rules. Afterwards these are applied to an adaptive video filter application. This paper closes with a conclusion and outlook on future work.
II. RELATED WORK

There are several design methodologies available that address the same demands as OSSS. SpecC [3] is completely refinement driven and introduces lots of useful concepts, like hierarchical behaviours and channels. Since it is based on the C language it does not fit well into the upcoming object-oriented development of SW dominated SoCs. Moreover, SystemC™ [4] has become IEEE standard most recently, which might boost its industrial acceptance. Furthermore, the upcoming OSCI TLM2 standard [5] is becoming very attractive for modelling communication architectures in complex SoCs. There have already been some attempts to use TLM in rapid prototyping [6], which are promising but lack automatic synthesis tools, like [7]–[9], allow behaviour level synthesis. However, these tools are well suited for synthesis of complex functional block, but lack adequate support for system synthesis. More general system level exploration frameworks like Metropolis [10] are very promising but mostly lack automatic synthesis support and cannot offer a seamless path to implementation. Work considering platform-based behaviour-level and system-level synthesis [11] aims at the same direction as our approach. The main difference to our approach is that we currently do not consider any behavioural synthesis and favour a user-driven scheduling for better traceability. Moreover, we focus more on the reuse and integration of existing IP components through mapping and transactor synthesis.

III. OSSS DESIGN METHODOLOGY

The following sections describe how the OSSS methodology can be used to develop embedded systems. For simplicity we only present a top-down design flow (see Fig. 1) that starts with a C++ “Golden Model”.

In a first step, this entry model is decomposed into structural elements (called behaviours). This is done systematically by profiling trials that identify relevant and computationally intensive elements. Profiling is enabled through model execution and manual trace point insertions. Generated traces can be analysed with the help of visualisation tools. After the identification and classification these structural elements are described by SystemC modules and OSSS Software Tasks. Communication and synchronisation between these parallel process elements is modelled by OSSS Shared Objects, that provide a method interface for communication and guarantee a consistent access of an arbitrary number of concurrent processes. More details of the sketched design flow are presented in Section IV.

This first structured system description in OSSS is called Application Layer Model (AL Model). The application is described in SystemC and C++ under consistent involvement of object-oriented features:

- Encapsulation of data and operations (methods) in classes. This is a basic object-oriented design principle for raising re-use.

- Method-based communication between structural blocks avoids effort of hand-crafted signal based communication and synchronisation. This kind of communication abstraction can be considered as high-level transaction level modelling (TLM), which is currently not covered by the OSCI TLM2 standard.

- Class inheritance allows easy extendibility for re-use. Polymorphism, which is based on inheritance relations, can be used to express implementation alternatives.

- Template classes offer easy parameterisation (e.g. buffer sizes) and make IP components more flexible.

- SystemC modules with ports and processes allow the specification of hardware components.

- OSSS Software Tasks specify parts of the application that are later executed on a processor.

On the AL we specify the function, logical structure, and an approximate time response of the system. Profiling results from analysis of the C++ “Golden Model” can be annotated to the AL to obtain an approximately timed behaviour. Also back-annotation approaches, where the execution of specific parts are profiled on their expected target technologies, are possible, but not further discussed here. Besides the functional correctness of the system, the AL offers an easy evaluation of design alternatives (e.g. HW/SW partitioning, scheduling, communication structures, and data locality). Profiling of different AL model alternatives with regard to their performance can be accomplished easily since the components allocations and scheduling can be changed quickly. Analysis of the executable AL model in early design phases can help to detect and resolve bottlenecks in the logical structure. This might result in the relocation of timely critical computations from SW to HW or in reorganising complex computations in pipeline structures to enhance the throughput.

The next step towards the implementation is the refinement to a so-called Virtual Target Architecture (VTA) Model. For this purpose more implementation details of the target architecture are added. Abstract communication relations from the AL model are mapped onto physical communication channels with different protocols and data bandwidths.
using different so-called OSSS-Channels [12] the designer can examine the influence of different bus protocols, data widths and arbitration schemes on the timing behaviour of the design. For saving costly communication resources, different logical communication relations can be grouped to a single physical connection. The resulting VTA model shows a cycle accurate timing behaviour and contains all other platform dependent information, like the chosen software processors or communication peripherals.

The VTA model is the input for the automatic synthesis process. *Fossy* (Functional Oldenburg System Synthesiser) generates synthesisable VHDL for each HW component. Furthermore, for the SW parts a driver API and for the HW parts bus interfaces are automatically generated. Depending on the chosen platform, different so-called architecture description files can be generated. Special properties of different target platforms require adoptions of the synthesis process, e.g. for embedding special IP blocks or the generation of 3rd party tool specific configuration files. Fig. 2 shows the *Fossy* synthesis process that has been tailored for Xilinx FPGA target technology.

IV. FROM APPLICATION TO IMPLEMENTATION

A. Model Composition

OSSS AL models can be composed of behaviours, Shared Objects and Shared Variables as shown in Fig. 3. Behaviours follow the Program-State Machine semantics [13] and can be hierarchically composed. They have their own thread of control and thus can be considered as active. They can be arranged in sequential, parallel or pipelined execution order. Each leaf behaviour consists of a functional core that is a passive entity object. In contrast to value objects, entity objects can not be assigned or copied. When the behaviour’s thread of control traverses this functional core it gets executed. A leaf behaviour can implement different alternatives of the same functional core. Communication of behaviours is performed by Shared Objects or Variables, whereas Shared Variables are specialised Shared Objects with no functional core and only a “single writer multiple reader” semantics. During HW/SW partitioning, behaviours are specialised to become either SW Tasks or HW Modules. This decision can be supported by different functional core implementations of the same behaviour. Consider one to be implemented in a Von-Neumann optimized, and the other one in a HW parallel style. Shared Variables can either be implemented by signals or they are bundled to shared memories. Until now Shared Objects always become specialised HW Modules. It is intended to provide a more flexible mapping to either HW or SW. Communication links constituted by the port-interface-bindings are mapped onto so-called OSSS Remote Method Invocation (RMI) channels [1]. They serve as protocol wrappers around physical communication resources like point-to-point channels and busses.

B. Adaptive Video Filter

Now the OSSS methodology is applied to the implementation of an adaptive video filter. Its basic operation is a discrete 2D convolution

$$c’(x, y) = \sum_{i=-\lfloor N/2 \rfloor}^{\lfloor N/2 \rfloor} \sum_{j=-\lfloor N/2 \rfloor}^{\lfloor N/2 \rfloor} h(i, j) \cdot c(x - i, y - j)$$

with filter matrix size $N \in \{x \in \mathbb{N} | x \ mod \ 2 \neq 0\}$, adaptive NxN filter kernel $h$, input image $c$, and filtered image $c’$. This filter operation should be applied to a video stream with a resolution of $720 \times 480$, 24 bit RGB, and a refresh rate of 60 Hz (DVD quality). For outputting this as a progressive picture (full image) a pixel clock of 27 MHz is required. This results in a data throughput of 648 MBit/s, which requires special HW support as offered by FPGAs or DSPs.

Fig. 4 illustrates the OSSS top-down design flow for the adaptive video filter design. We start with an algorithmic specification of the filter chain as initial C++ functional model (see Fig. 4(a)). In a first step the RGB colour space is reduced to luminance (gray scale). Before executing the NxN filter operator we need to save $N$ video lines into a line buffer. The filtered pixel is written to a video stream sink afterwards.

We propose the use of classes for modelling both, value and behaviour types. Listing 1 shows a code snippet of the functional NxN filter implementation. The MAC (multiply accumulate) operation of the 2D convolution is implemented as an overloaded operator of the Filter_Matrix template.
class (see line 1-18). It is derived from a square matrix template class, which itself is implemented as a vector of vectors (see line 3). The `operator*` gets a coefficient matrix and performs the element-wise MAC operation.

```cpp
template<class T, unsigned int dimension>
struct Filter_Matrix :
{
  public:
    Vector<T, dimension>, dimension>
    
    template<typename Matrix_t> Matrix_t
    operator*(const Filter_Matrix<Matrix_t, dimension>
              &matrix) const
    {
      Matrix_t result = 0;
      for (unsigned int i=0; i<dimension; ++i)
        for (unsigned int j=0; j<dimension; ++j)
          Matrix_t coeff = matrix.get(i, j);
          T pixel = this->get(i, j);
          result += coeff * pixel;
    }
    return result;
}

template<typename pixel_t, typename coeff_t, 
unsigned int N>
class NxN_Filter {
  public:
    coeff_t operator() (const Vector<pixel_t, N> &data) {
      pixel_matrix <<= 1;
      pixel_matrix.setCol(0, data);
      coeff_t result = pixel_matrix * coeff_matrix;
      return (result >> Filter::COEFF_PRECISION) + shift;
    }

  protected:
    coeff_t shift;
    Filter_Matrix<pixel_t, N> coeff_matrix;
    Filter MATRIX<coeff_t, N> coeff_matrix;
};
```

Listing 1. NxN Filter Class

```cpp
template<class Beh_t = osss behaviour>
class NxN_Filter_Beh : public Beh_t {
  public:
    osss_in< Vector<Image::pixel_t, Filter::MATRIX_DIM >>
    pixel_row;
    osss_out< Filter::coeff_t > filtered_pixel;
    osss_port< osss_shared_if<nxn_config_read_if > >
    nxn_conf;

    void config() {
      filter_conf = nxn_conf->get();
      wait();
    }

    void filter() {
      filter.set_coeffs(filter_conf);
      filtered_pixel = filter(pixel_row);
      wait();
    }

  protected
    NxN_Filter<Image::pixel_t, Filter::coeff_t, 
              Filter::MATRIX_DIM> filter;
    NxN_Configuration<Image::pixel_t, Filter::coeff_t, 
                     Filter::MATRIX_DIM> filter;
};
```

Listing 2. NxN Filter Behaviour

While the `Filter_MATRIX` class defines a value type, the `NxN_FILTER` class functor defines an entity type (see line 20-30), that is non-copyable and non-assignable. The `operator()` feeds a new line into the pixel matrix (left shift by one and column assignment), calls `operator*` from
Filter_Matrix, performs a normalisation and returns the filtered pixel.

The AL model of the video filter design in Fig. 4(b) is used to explore the possible parallelism from the purely sequential C++ functional specification. The video filter chain is best implemented as a pipeline with the filter selector behaviour running in parallel. The communication between the pipeline stages is performed by Shared Variables. Communication between the filter selector and the NxN filter is modelled using a Shared Object. Listing 2 illustrates the implementation of the NxN filter behaviour. It contains a port for an incoming pixel vector (line 4-5), an output port for the filtered pixel (line 6), and a port to the filter configuration Shared Object (line 7-8). The config process gets filter configurations from the Shared Object and writes them to a local member (line 11). The filter process runs in parallel (line 21) to the config process and sets the filter configuration at the NxN_Filter functional core (line 16) from Listing 1. The filter operation itself is performed by calling the operator() of NxN_Filter at line 17. At this point we would like to emphasise that the functional behaviour from the C++ specification is completely reused as functional cores.

The final step to bring the AL model towards the VTA (see Fig. 4(c)) is finding a suitable HW/SW partitioning. For this example we have decided to perform the filter chain in HW and the adaptive filter coefficient calculation and selection in SW. Therefore, the filter selector behaviour is converted into a Software Task and mapped onto a Xilinx MicroBlaze™ soft-core processor. The NxN filter configuration Shared Object is attached to the MicroBlaze’s On-Chip Peripheral Bus (OPB). The communication from the software task to the Shared Object is implemented using the OSSS RMI Technology. No further manual communication refinement is required to make the step from AL to VTA layer.

For synthesis reasons we have manually converted all behaviour blocks to be implemented in HW to sc_modules and all shared variables between them to sc_signals. This step should become obsolete when we support the immediate synthesis of behaviours with Fossy.

Furthermore, the video line buffer is refined to make use of a Xilinx Block RAM (BRAM) resource. The mapping of large arrays to special memory resources will be performed automatically in the future. Nevertheless, we support the seamless integration of common RTL IP components with even more sophisticated protocols and signal level interfaces. This is demonstrated by the interface refinement of the RGB to luminance and the NxN filter to a signal level protocol of the external de-interlacer and video DAC IPs.

The VTA layer model is then automatically processed to a target technology dependent implementation using the synthesis flow from Fig. 2. As shown in Fig. 4(d) we have placed the synthesised video filter chain into a video processing environment on a Virtex-II Pro FPGA equipped with a video decoder daughter board.

V. Conclusion

In this paper we have presented the OSSS methodology for embedded HW/SW systems and highlighted the following features: OSSS enables design and system synthesis of C++/SystemC models in a homogeneous system-level description language and simulation environment. We support a stepwise and seamless refinement of HW and SW components and allow easy integration of pre-existing IP components. This ranges from RTL IP components with a cycle accurate pin interface, like RAMs, to system-level IP components, like SW processors and buses. In our platform-based approach AL models allow the abstract modelling of the design and the exploration of different logical structures and system-level scheduling. VTA models describe the physical properties and allow the comparison of alternative implementations on different platforms. Finally, an automatic synthesis with Fossy, which supports the SystemC synthesisable subset as well as manifold system-level modelling concepts (like Shared Objects, Software Tasks, and OSSS-Channels) for raising the designers productivity, transforms the VTA model to an implementation. The next steps involve the synthesis of more abstract process control features like pipeline behaviours, and a more orthogonal communication modelling with Shared Objects and Shared Variables.

References