Simulation-Based Design Methodology for Heterogeneous Systems at Package-Level Utilizing XML and XSLT

Robert Fischbach[a,b], Andy Heinig[a], and Jens Lienig[b]
[a]Fraunhofer Institute for Integrated Circuits, Division Engineering of Adaptive Systems, Dresden, Germany
[b]TU Dresden, Institute of Electromechanical and Electronic Design, Dresden, Germany

Abstract

System-in-Package is an appealing alternative compared to the integration on a PCB or in a chip. A big variety of different packaging solutions (including 2.5/3D integration) makes it difficult to choose the most appropriate solution for a given specification. Simulation-based design flows gain importance, but lack the straightforward access to actual design data. We propose a new description format as well as a corresponding methodology to manage and process assembly and packaging design data. Based on established software concepts (XML/XSLT), our Assembly Description Format (ADF) integrates well into existing design environments and features a high flexibility to consider distinct design aspects.

1 Introduction

The growing number of heterogeneous components integrated into single system nodes is driven by applications like the Internet of Things (e.g., multi sensor integration), autonomous driving (e.g., data fusion), and cyber-physical systems (e.g., bio-signal processing). To successfully compete on these markets, increasing functionality, shrinking dimensions, and/or higher computational performance are essential. Thus, a tight integration of sensors, analog, and digital circuits is needed.

Historically, the required complexity was integrated into chips and assembled on a circuit board. Recently, a trend towards wafer- and package-level integration can be observed, as these approaches offer multiple advantages [1]. For example, high pin-count components require finer interconnect structures compared to PCB, and different semiconductor technologies can be combined economically on package-level.

However, package-level integration lacks common design knowledge and proven design methodologies, which are a prerequisite to handle challenges like chip/package/board co-design as well as the big variety in packaging options. With advanced system integration, such as interposer-based 2.5/3D integration, cost estimations become more relevant (compared to the costs of classical packages, like Dual In-line (DIP) or Quad Flat Packages (QFP)).

The development of integrated systems on package-level typically requires **co-design flows** able to combine heterogeneous EDA (Electronic Design Automation) software tools to address the different steps on chip-, package-, and board-level. The few number of available publications detailing the co-design flow usually originate from larger semiconductor companies, such as Intel, IBM, or Infineon. A sophisticated interaction between the multiple software tools and file formats of different EDA vendors (e.g., Cadence, Synopsys, Mentor) is the common message [2, 3, 4].

Typical design steps are: top-level netlist handling, component management (e.g., footprints, symbols, substrate assignment), co-optimization, and co-verification. Current academic research related to package-level system design mainly focuses on physical design algorithms, such as 2.5/3D floorplanning, routing, and placement [5, 6].

Understandable but unfortunate, every company fosters its own individual tool landscape and corresponding co-design flows, as they comprise a company's valuable expertise. A flexible generic approach to chip/package/board co-design, which could be adopted by smaller companies, hardly emerges in this environment. Furthermore, the electronics industry would greatly benefit from new design tools (e.g., for package-level design-space-exploration, knowledge-based system engineering). Hence, our paper presents a methodology to model heterogeneous 3D-integrated systems on package-level and to automatically derive data required for additional design flow steps, such as simulation.

With the increasing relevance of simulation-based design approaches, such a methodology can be applied to explore designs without time-consuming prototypes and design cycles. This enables the so far missing comparisons of multiple packaging variants. Sub-models with different levels of detail allow choosing between fast explorations and validated sign-off investigations (e.g., design rule checks). The formal system model also enables the export of manufacturing data [7].

The presented methodology is based on the established software technology XML/XSLT, which makes it possible to use mature editors, transformation engines, and scheme validation tools. The XML/XSLT origin facilitates an integration into modern end-user software as many software-development concepts integrate well with XML data.

After the description of the design example used throughout the paper in **Section 1.1**, **Section 2** explains the Assembly Description Format (ADF) (with a focus on 3D ge-

Please quote as: R. Fischbach, A. Heinig, J. Lienig "Simulation-Based Design Methodology for Heterogeneous Systems at Package-Level Utilizing XML and XSLT," GMM-Fachbericht 274, Reliability by Design Conf. (ZuE 2017), VDE Verlag, ISBN 978-3-8007-4444-2, pp. 66-73, Sept. 2017.

 © VDE VERLAG GMBH · Berlin · Offenbach

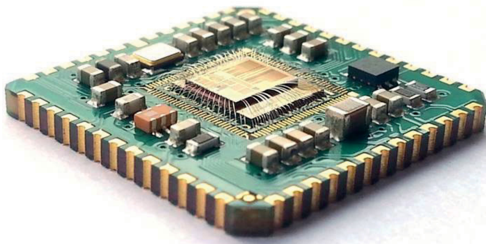

Figure 1 Design example: biomedical SiP (LCC package) with a wire-bonded stack of two bare dies.

ometries) developed to represent 3D-integrated systems on package-level as well as the basic concepts of XSLT. **Section 3** contains the transformation steps from input data processing, over applying package model modifications, towards the export into simulation. The integration of the presented methodology into existing design flows is illustrated in **Section 4**. Finally, **Section 5** summarizes this paper and provides an outlook to future work.

1.1 Design example

Figure 1 shows a picture of the package which is used as a design example throughout this paper. The depicted package type is a Leadless Chip Carrier with 52 terminals (LCC52). This LCC houses a biomedical System-in-Package (SiP) which integrates two bare dies with additional components, such as decoupling capacitors. The LCC was designed as a modular component for the usage within a larger biosignal sensing platform.

For the modeling in this paper we will focus on the die stack in the middle of the LCC package, where two bare dies are wire-bonded onto the package substrate. The first die (bottom) is a Texas Instruments (TI) ADS1299; a low-noise, 8-channel, 24-bit analog-to-digital converter (ADC) for electromyography (EMG), electrocardiography (ECG), and extracranial electroencephalogram (EEG) applications. The second die (top) is a Microchip (former Atmel) ATmega256RFR2; a System-on-Chip (SoC) combining a 16 MHz, 256 kilobyte flash, 8-bit microcontroller, and a 2.4GHz RF transceiver. The electrical connections between the die pads and the bond fingers located on the package substrate are established by thermosonic bonding using 25 µm gold bond wires.

1.2 Tool setup

The transformation steps described in **Section 3** were tested under Linux and Windows. Based on Java SE 8 Update 121, we applied Xalan-Java Version 2.7.1 to process the XML/XSLT files on both platforms [8]. Xalan-Java is open source and contains basic XSLT extensions (EXSLT), such as trigonometric functions used for the geometrical transformations within our methodology (e.g., to apply rotations to components). The editing of XML and XSLT files can be accomplished with every standard text editor. In addition, schema validation and stylesheet debugging are valuable features provided by dedicated XML tools or plugins. As described in **Section 3.3**, the CAD data was generated using the open source CAD platform Open Cascade [9]. Finally, the created design data was tested in COMSOL Multiphysics version 5.1 [10].

2 XML/XSLT fundamentals

This section explains the basics of the Assembly Description Format (ADF), which evolved from our early XML-based hierarchical description of 3D systems and SiP [11]. XML stands for Extensible Markup Language and enables a structured representation of hierarchical data, which is both human- and machine-readable [12]. The wide distribution of XML and the comprehensive ecosystem (e.g., tooling, related languages) makes it a good choice for data exchange and domain specific languages.

A flexible description of a package is only one aspect required in a design flow, the transition steps towards the desired package layout is another. To modify a model represented in ADF, Extensible Stylesheet Language Transformations (XSLT) are applied. XSLT is a Turing-complete XML-based language and belongs to a family of W3C recommendations for defining XML document transformations [13]. A condensed summary of the basic concepts used throughout this paper can be found below.

2.1 XML-based description of packages using the Assembly Description Format

The current implementation of ADF and the corresponding software tools mainly focus on the geometric representation of integrated systems on package-level. Nevertheless, ADF is extensible with language elements to describe various additional aspects of the assembly process in general (e.g., electrical connectivity, assembly rules, process steps). With this flexibility it is an appropriate solution to implement Assembly Design Kits (ADK). The ADK concept targets to be of comparable importance in the packaging and assembly field as the Process Design Kit (PDK) is for chip design [7].

In ADF, the geometric representation is based on the concept of Constructive Solid Geometry (CSG). In CSG complex solids are built from primitive basic elements and their combination using Boolean operations [14]. **Table 1** lists the geometric elements of ADF and the corresponding attributes. The root of an ADF file is the *library* element containing documentation (*author*, *documentation*, *history*) and *module* elements. A module is the central element to define a single component and can be instantiated in other modules to create a hierarchical structure. Within a module, the CSG-like geometry description is initiated with the first occurrence of a *compound* or *unite* element, respectively. The grouping (*compound*), Boolean topological operations (*unite*, *intersect*, and *cut*), as well as the transformations (*translate* and *rotate*) are non-primitives and can be nested without restrictions. Non-primitives can also contain the primitive elements (*cuboid*, *sphere*, *cylinder*, *include*), which create basic geometric solids or in-

ISBN 978-3-8007-4444-2

© VDE VERLAG GMBH · Berlin · Offenbach

Table 1 Excerpt of basic **geometrical** elements of the XML-based Assembly Description Format.

Element	Attributes	Description
library	name, version	root element, contains *author*, *documentation*, *history*, and *module*
module	name	defines a component containing a single *unite* or *compound* element
compound		defines a loosely combined set of elements (e.g., used for grouping)
unite		(geometrically) fuses the contained elements into a single element
intersect		(geometrically) intersects the contained elements into a single element
cut		(geometrically) cuts the second and following elements from the first
translate	x, y, z	shifts the contained elements along the given vector
rotate	x, y, z	rotates the contained elements around the x-, y-, and z-axis
cuboid	name, length, width, height	defines a primitive cuboid, origin at its center
sphere	name, radius	defines a primitive sphere, origin at its center
cylinder	name, radius, length	defines a primitive cylinder, origin at the center of its base
include	name, module	instantiates a previously defined module

stantiate more complex geometries previously defined in other modules, respectively.

Please note that Table 1 is only a snapshot of the current ADF, as it is continuously extended by additional elements (e.g., half spaces, shape sweeping). Despite the low number of different elements, the ADF is able to model sophisticated geometries due to the underlying CSG concept. Beyond the plain CSG concept, we plan to integrate control structures known from programming languages (e.g., conditionals, loops) to enable a more efficient modeling (e.g., to create arrays of components).

Listing 1 gives a basic example of an ADF file. This non-hierarchical example comprises only a single “Top” module to compound two cuboids (a “substrate” and a “die”) and three balls. The *translation* elements cause the correct positioning of the five primitive solids.

2.2 XSLT concepts used in this paper

XSLT defines elements to describe the transformation of an XML input file. The central element is the *template*. It is used to match a specified path in the input document or to define a reusable “sub-routine”, respectively. Once a template matches, further logic elements (well-known from typical programming languages, e.g., *if*, *for-each*, *variable*) are used to define a transformation. Expressions in these elements are used to manipulate data and can contain XPath syntax to navigate inside the XML document tree.

To illustrate the usage of XSLT to modify a given ADF representation, we will extend the previous example by adding pad structures to the die and the substrate. **Listing 2** shows the short XSLT which implements this task. On line 8 begins a “copy”-template which recursively matches everything and outputs a copy of the current match. If this file would not define further templates, the output of the transformation would be a copy of the input file.

Listing 1 Basic ADF example representing the geometries shown in Figure 2 (left).

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <library
3      xmlns="http://eas.iis.fraunhofer.de/electr"
4      xmlns:g="http://eas.iis.fraunhofer.de/geom"
5      version="2017-05-05" name="example">
6    <author> robert.fischbach@tu-dresden.de </author>
7    <documentation> Example </documentation>
8    <g:module name="Top">
9      <g:compound>
10       <g:cuboid name="substrate" length="2000"
11                 width="2000" height="400"/>
12       <g:translate x="0" y="0" z="275">
13         <g:translate x="-300" y="0" z="0">
14           <g:sphere name="ball_1" radius="75"/>
15         </g:translate>
16         <g:sphere name="ball_2" radius="75"/>
17         <g:translate x="300" y="0" z="0">
18           <g:sphere name="ball_3" radius="75"/>
19         </g:translate>
20       </g:translate>
21       <g:translate x="0" y="0" z="450">
22         <g:cuboid name="die" length="1000"
23                   width="1000" height="200"/>
24       </g:translate>
25     </g:compound>
26   </g:module>
27 </library>
```

Listing 2 XSLT file to extend the basic ADF example from Listing 1 with die and substrate pads.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <xsl:stylesheet
3      xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
4      xmlns="http://eas.iis.fraunhofer.de/electr"
5      xmlns:g="http://eas.iis.fraunhofer.de/geom"
6      version="1.0">
7    <xsl:output method="xml" indent="yes"/>
8    <xsl:template match="/ | @* | node()">
9      <xsl:copy>
10       <xsl:apply-templates select="* | @* | node()"/>
11     </xsl:copy>
12   </xsl:template>
13   <xsl:template
14      match="//g:sphere[starts-with(@name, 'ball_')]">
15     <xsl:variable name="index"
16        select="substring-after(@name, 'ball_')"/>
17     <g:unite>
18       <g:translate x="0" y="0" z="{@radius - 5}">
19         <g:cuboid name="{concat('die_pad_', $index)}"
20             length="150" width="150" height="10"/>
21       </g:translate>
22       <xsl:copy>
23         <xsl:apply-templates
24             select="* | @* | node()"/>
25       </xsl:copy>
26       <g:translate x="0" y="0" z="{5 - @radius}">
27         <g:cuboid
28             name="{concat('substrate_pad_', $index)}"
29             length="150" width="150" height="10"/>
30       </g:translate>
31     </g:unite>
32   </xsl:template>
33 </xsl:stylesheet>
```

Line 13 starts the second template defined in this file. As soon as a more specific match is defined, the XSLT processor will prefer it. Here, the template matches every *sphere*

Figure 2 Geometric visualization of the transformation. ADF example without pads (left, compare Listing 1) and with pads (right, compare Listing 3).

element where the attribute “name” contains a string starting with “ball_”. Applied to the input file, the second template matches the *sphere* elements on lines 14, 16, and 18 in Listing 1.

Next, a variable named “index” is defined to extract the index number encoded in the name attribute of the *sphere* elements. Within an XSLT expression the @-sign is used to access element attribute values and the $-sign is used to access variables. “starts-with()” is one of many available XPath functions that can be used within XSLT expressions. As a result, this variable will contain the value 1, 2, or 3, depending on the actual element currently matched.

The following additional ADF elements are send to the output file to add the aforementioned pad structures. The first *translate* element shifts the inserted die pad (new *cuboid* element) upwards. Subsequently, the original *sphere* element is copied to the output before a second *translate* and *cuboid* element create the substrate pad.

Xalan-Java is used to apply the XSLT file (Listing 2) to the ADF example (Listing 1). With the Java CLASSPATH correctly set, this command line call will perform the transformation:

```
java org.apache.xalan.xslt.Process
    -IN example.xml
    -XSL example_transform.xsl
    -OUT transform_result.xml
```

The result of the transformation is shown in **Listing 3** and **Figure 2** (right) which exhibit the added pads.

Listing 3 Result (excerpt) of the transformation in Listing 2 applied to the basic ADF example in Listing 1.

```
1  <g:module name="Top">
2    <g:compound>
3      <g:cuboid name="substrate" length="2000"
4               width="2000" height="400"/>
5      <g:translate x="0" y="0" z="275">
6        <g:translate x="-300" y="0" z="0">
7          <g:unite>
8            <g:translate z="70" y="0" x="0">
9              <g:cuboid height="10" width="150"
10               length="150" name="die_pad_1"/>
11           </g:translate>
12           <g:sphere name="ball_1" radius="75"/>
13           <g:translate z="-70" y="0" x="0">
14             <g:cuboid height="10" width="150"
15               length="150" name="substrate_pad_1"/>
16           </g:translate>
17         </g:unite>
18       </g:translate>
```

3 Transformation steps

The ability to alter a package represented in ADF is the prerequisite to use it within a design process. After importing the input data to ADF, we will show how to apply modifications to the package model. Finally, the data is exported into a CAD format to be readable by established 3rd party tools, such as the COMSOL simulation environment.

Applying non-robust or partial transformations can result in invalid physical configurations. ADF is not limited in this regard to maintain a high flexibility. However, input data validation and the verification of the resulting data helps mitigating this issue. Validation on CSG based geometries is well investigated and for example easily allows to test against collisions or overlaps. Comparable to Design Rule Checks (DRC) in IC design flows, design rules can also be applied to the 3D geometries modeled by ADF [15].

The order of the transformations described next is relevant and dictated by the design flow at a higher level. Nevertheless, this is not a general limitation of ADF. Carefully implemented transformations can be applied multiple times during a design flow sequence during different steps (e.g., refinement of placeholder structures).

3.1 Processing of input data

With the focus on the die stack, our design example shown in Figure 1 comes with the following specifications.

die geometries comprise the die’s dimensions and orientation within the stack as well as the positions of the die pads; this information is typically extracted from data sheets

bond finger geometries comprise the position and the dimensions of the bond fingers located on the substrate; these geometries are based on design rules provided by an assembly house

electrical connectivity comprises the required signal mapping from the die pads towards specific bond fingers; this information is usually derived from a higher level (e.g. package-level) netlist

3.1.1 Data preparation

In general, the input data is available in textual or tabular form (very common are txt-, csv- (comma-separated values), or spreadsheet-files). A translation into XML is necessary to use this data inside our suggested XML/XSLT-based methodology. This step is straightforward and can be achieved in various ways. For example, short scripts can be used to put the required values into an XML structure, common spreadsheet file formats (e.g., xslx) are already XML, and many established design tools allow a script-based output of internal data into user specified file formats. The definition of an XML input format even features additional advantages, such as the possibility to define a schema used for syntax validation and a good support by third party tools. The result of the prepared input data is shown in **Listings 4, 5, and 6**[1]

[1]Electrical connectivity is discussed in Section 3.2.

Listing 4 Input XML description of the die stack. Length units are given in µm and angles in degrees.

```
<?xml version="1.0"?>
<diestack alignment="centered">
  <die name="ATmega256RFR2" xDim="4855" yDim="3865"
     zDim="100" zOffset="500" padXDim="70"
     padYDim="70" padPosFile="esimed_chip_pads.xml"/>
  <die name="TI1299" xDim="6005" yDim="5585"
     zDim="100" zOffset="400"
     rotation="-90" padXDim="80"
     padYDim="80" padPosFile="esimed_chip_pads.xml"/>
  <substrate name="LCC_Substrate" xDim="8300"
     yDim="8600" zDim="400" zOffset="0"
     bondfingerPosFile="esimed_bondfinger.xml"/>
</diestack>
```

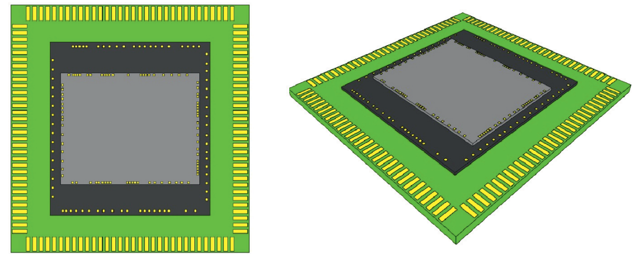

Figure 3 Graphical visualization of the imported die stack (left: top view, right: axonometric view).

Listing 5 Input XML with bond finger dimensions and positions located on the substrate (excerpt).

```
<bondfingers>
  <bondfinger name="finger_1" length="500"
              width="130" x="-3850.0" y="3565.0"/>
  <bondfinger name="finger_2" length="500"
              width="130" x="-3850.0" y="3335.0"/>
  <!-- ... -->
  <bondfinger name="finger_128" length="130"
              width="500" x="-3565.0" y="4000.0"/>
</bondfingers>
```

Listing 6 Input XML to describe the pad positions of both stacked dies (excerpt).

```
<diepads>
  <diepad die="ATmega256RFR2" name="1_PF2:ADC2:DIG2"
        x="-2362.5" y="1517.5"/>
  <diepad die="ATmega256RFR2" name="2_PF3:ADC3:DIG4"
        x="-2362.5" y="1373.5"/>
  <!-- ... -->
  <diepad die="TI1299" name="1_IN8N"
        x="-2382.5" y="-2692.5"/>
  <diepad die="TI1299" name="2_IN8P"
        x="-2062.5" y="-2692.5"/>
  <!-- ... -->
</diepads>
```

3.1.2 Input data transformation

With the input data properly prepared, a first transformation is needed to read the required information into the ADF. **Listing 7** shows an excerpt of the applied XSLT. Three templates are shown. The first template on line 1 matches the root element of the chip stack file (cf. Listing 4), the second template matches a specific chip within this chip stack, and the third template matches the corresponding die pads taken from the die pads file (cf. Listing 6). This combination of several input files is possible by using the XPath function "document()" on line 32, which offers a high flexibility to integrate multiple data sources into the ADF. The *if* elements on lines 24 and 42 demonstrate the possibility to conditionally branch within the XSLT file, for example, depending on a parameter passed into the template.

Listing 7 Excerpt of the XSLT file used to import the design data of the several XML input files into an ADF representation.

```
<xsl:template match="/chipstack">
  <library
     xmlns="http://eas.iis.fraunhofer.de/electr"
     xmlns:g="http://eas.iis.fraunhofer.de/geom"
     name="INPUT2EAS example" version="2017-05-04">
    <author> robert.fischbach@tu-dresden.de </author>
    <documentation> Test design </documentation>
    <xsl:apply-templates>
      <xsl:with-param name="context">module_creation
      </xsl:with-param>
    </xsl:apply-templates>
    <g:module name="Top">
      <g:compound>
        <xsl:apply-templates>
          <xsl:with-param name="context">
                instance_call_creation
          </xsl:with-param>
        </xsl:apply-templates>
      </g:compound>
    </g:module>
  </library>
</xsl:template>
<xsl:template match="chip">
  <xsl:param name="context"/>
  <xsl:if test="$context = 'module_creation'">
    <g:module name="{@name}">
      <g:compound>
        <g:translate x="0" y="0" z="{@zDim div 2}">
          <g:cuboid name="die" length="{@xDim}"
             width="{@yDim}" height="{@zDim}"/>
        </g:translate>
        <g:translate x="0" y="0" z="{@zDim}">
          <xsl:apply-templates select="document(@padPosFile)/
                diepads/diepad[@die=current()/@name]">
            <xsl:with-param name="padXDim"
                select="@padXDim"/>
            <xsl:with-param name="padYDim"
                select="@padYDim"/>
          </xsl:apply-templates>
        </g:translate>
      </g:compound>
    </g:module>
  </xsl:if>
  <xsl:if test="$context = 'instance_call_creation'">
   <g:translate x="0" y="0" z="{@zOffset}">
      <xsl:choose>
        <xsl:when test="@rotation">
          <g:rotate x="0" y="0" z="{@rotation}">
            <g:include module="{@name}"
                name="{@name}_Inst"/>
          </g:rotate>
        </xsl:when>
        <xsl:otherwise>
          <g:include module="{@name}"
             name="{@name}_Inst"/>
        </xsl:otherwise>
      </xsl:choose>
    </g:translate>
  </xsl:if>
</xsl:template>
<xsl:template match="diepad">
  <xsl:param name="padXDim"/>
  <xsl:param name="padYDim"/>
  <g:translate x="{@x}" y="{@y}" z="2.5">
    <g:cuboid name="diepad_{@name}"
       length="{$padXDim}" width="{$padYDim}"
       height="5"/>
  </g:translate>
</xsl:template>
```

The script combines the provided information of the three input XML files to generate a geometric representation of the die stack. The visualization of the resulting ADF file is shown in **Figure 3**.

The processing of the input data as well as the following transformation steps will be put into context in **Section 4**.

ISBN 978-3-8007-4444-2

© VDE VERLAG GMBH · Berlin · Offenbach

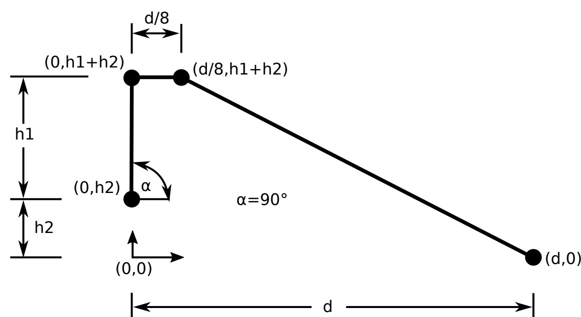

Figure 4 Simplified bond wire model in accordance with the EIA/JEDEC bond wire modeling standard [16]. The model parameters are: h1 - the height between the bond pad and the top of the loop, h2 - the height between the bond pad and the lead, and d - the total distance the wire covers in the horizontal plane.

3.2 Applying package model modifications

Now, that an ADF representation of the substrate and the dies exists, the initial package model evolves towards a manufacturable design. Depending on the intended purpose of the package model, various validated sub-models can be applied. For example, one sub-model could be validated for thermal simulation and another for the export of manufacturing data under the consideration of given assembly rules. This approach can concern all assembly and design aspects, such as the molding of bare dies or actual substrate metal layers.

In our example, we will consider the electrical connectivity between the die pads and the bond fingers by adding wire bonds to the die stack as described in **Section 1.1**. The required signal mapping is part of the design netlist and (just like the geometrical specification in the previous section) was transformed into a suitable XML format (see **Listing 8**). Each *mapping* element within this file defines a connection between a specific die pad and a bond finger.

Listing 8 Input XML file representing the electrical connectivity mapping between the die pads and the bond fingers (excerpt).

```
1  <mappings>
2    <mapping dieName="TI1299"
3            diePadNumber="1" bondfingerNumber="1"/>
4    <mapping dieName="TI1299"
5            diePadNumber="2" bondfingerNumber="2"/>
6    <!-- ... -->
7    <mapping dieName="ATmega256RFR2"
8            diePadNumber="1" bondfingerNumber="3"/>
9    <mapping dieName="ATmega256RFR2"
10           diePadNumber="2" bondfingerNumber="5"/>
11   <!-- ... -->
12 </mappings>
```

3.2.1 Bond wire models

Depending on the required model accuracy, different bond wire models are available [15]. We focus on the EIA/JEDEC bond wire modeling standard which includes two models with different complexities [16].

Figure 4 shows the simplified bond wire model used in our example. It only needs three parameters to represent a bond wire. Additionally, the standard also defines a preferred model, which considers two additional parameters in order to reduce the deviation from the exact bond wire trajectory.

3.2.2 Insertion of bond wires

In our design, the bond wires always start at a die pad and end on a bond finger. The correct mapping is taken from the aforementioned mapping file (see Listing 8). **Listing 9** demonstrates how a simplified bond wire can be inserted via an XSLT transformation applied to the ADF representation of the package design.

Listing 9 Excerpt of the XSLT file used to insert a simplified bond wire into the package design.

```
1  <xsl:template name="GenerateSimpleWirebond">
2    <xsl:param name="name"/>
3    <xsl:param name="x1"/>
4    <xsl:param name="y1"/>
5    <xsl:param name="z1"/>
6    <xsl:param name="x2"/>
7    <xsl:param name="y2"/>
8    <xsl:param name="z2"/>
9    <xsl:param name="loopheight" select="250.0"/>
10   <xsl:param name="diameter" select="25.0"/>
11   <xsl:variable name="radius"
12     select="$diameter div 2"/>
13   <xsl:variable name="h1" select="$loopheight"/>
14   <xsl:variable name="h2" select="$z1 - $z2"/>
15   <xsl:variable name="d" select="math:sqrt(math:power($x2 -
        $x1, 2) + math:power($y2 - $y1, 2)) + $offset"/>
16   <xsl:variable name="seg3Len"
        select="math:sqrt(math:power($d - $d div 8, 2) +
        math:power($h1 + $h2, 2))"/>
17   <xsl:variable name="seg3Ang" select="180 * math:atan2($h1
        + $h2, $d - $d div 8) div math:constant('PI', 12)"/>
18   <xsl:variable name="rotation" select="180 *
        math:atan2($y2 - $y1, $x2 - $x1) div
        math:constant('PI', 12)"/>
19   <g:translate x="{$x1}" y="{$y1}" z="{$z1}">
20     <g:rotate x="0" y="0" z="{$rotation}">
21       <!-- segment 1 -->
22       <g:cylinder name="{concat($name,'_segment_1')}"
23           height="{$h1}" radius="{$radius}"/>
24       <!-- segment 2 -->
25       <g:translate x="0" y="0" z="{$h1}">
26         <g:rotate x="0" y="90" z="0">
27           <g:sphere
28               name="{concat($name, '_sphere_1_2')}"
29               radius="{$radius}"/>
30           <g:cylinder
31               name="{concat($name, '_segment_2')}"
32               height="{$d div 8}"
33               radius="{$radius}"/>
34         </g:rotate>
35       </g:translate>
36       <!-- segment 3 -->
37       <g:translate x="{$d div 8}" y="0" z="{$h1}">
38         <g:rotate x="0" y="{$seg3Ang + 90}" z="0">
39           <g:sphere
40               name="{concat($name, '_sphere_2_3')}"
41               radius="{$radius}"/>
42           <g:cylinder
43               name="{concat($name, '_segment_3')}"
44               height="{$seg3Len}"
45               radius="{$radius}"/>
46         </g:rotate>
47       </g:translate>
48     </g:rotate>
49   </g:translate>
50 </xsl:template>
```

The corresponding template is called with parameters for the naming, the start and end position, and (optionally) the loop height as well as the wire bond diameter. Lines 13-15 contain the calculation of h1, h2, and d (according to the EIA/JEDEC model). The remaining statements generate the actual ADF elements for the three bond wire segments using the XSLT math extension. The result of the bond wire insertion is visible in **Figure 5**.

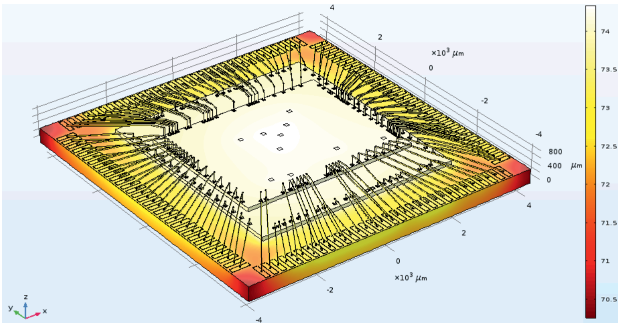

Figure 5 Thermal simulation of the wire-bonded die stack with COMSOL.

3.3 Export to simulation

A meaningful simulation requires a realistic setup and a sufficiently detailed model of the object. This includes issues like the proper setting of boundary conditions in the simulation environment, the availability of validated material properties, and exact but manageable object geometries (i.e., the object can be meshed and simulated in reasonable time). Some of our previous works already address these challenges [17, 18]. The work in this paper focuses on the (automatic) geometry generation based on real layouts by combining existing data from design tools, files, and specifications.

In order to provide the package geometry in a CAD file format supported by simulation tools, we chose an indirect export as follows. First, we applied an XSL transformation from ADF into a script for Open Cascade. Second, we run this script file in Open Cascade to generate a suitable CAD file (e.g., STEP). Another (less flexible) option would have been to directly write a STEP file via XSLT.

Listing 10 Excerpt of the XSLT file used to transform an ADF representation into an Open Cascade script to generate a CAD format file.

```
1  <xsl:output method="text" indent="no"/>
2  <xsl:template match="/e:library">
3    <xsl:if test="(local-name(g:module[@name =
           'Top'][1]/*[1]) = 'unite') or
           (local-name(g:module[@name = 'Top'][1]/*[1]) =
           'compound')">
4      <xsl:value-of
5        select="concat('pload MODELING', $newline)"/>
6      <xsl:value-of
7        select="concat('pload OCAF', $newline)"/>
8      <xsl:value-of
9        select="concat('pload XDE', $newline)"/>
10     <xsl:apply-templates
11       select="g:module[@name = 'Top'][1]"/>
12     <xsl:variable name="top"
13       select="concat('Top_', local-name(g:module[@name =
           'Top']/*[1]), '_1')"/>
14     <xsl:value-of
15      select="concat('NewDocument D XCAF',$newline)"/>
16     <xsl:value-of
17       select="concat('XAddShape D ',$top,$newline)"/>
18     <xsl:value-of
19       select="concat('WriteStep D ', '&quot;', $top,
             '.step&quot;', $newline)"/>
20   </xsl:if>
21 </xsl:template>
22 <xsl:template match="g:sphere">
23   <xsl:param name="parent_name"/>
24   <xsl:variable name="name"
25     select="concat($parent_name,'_sphere_',@name)"/>
26   <xsl:value-of
27     select="concat('psphere ',$name,' ',@radius,$newline)"/>
28 </xsl:template>
```

Listing 10 shows an excerpt of the output transformation file. It demonstrates two new aspects. First, an XSLT can output to a plain text file, too (see line 1). Second, this output is typically realized by writing out values (using the *value-of* element). Line 19 contains the actual command, which finally triggers the writing of the STEP file. Set up correctly, Open Cascade executes the generated script with the following command:

```
draw -f oc_script.tcl
```

As a result, the desired CAD file is generated, which can be imported into a simulation environment like COMSOL. Figure 5 shows the result of the thermal simulation applied to the wire-bonded die stack. Depending on the simulation requirements, the ADF-based methodology enables the selection of adjusted sub-models (e.g., simulation speed vs. accuracy). The material properties and boundary conditions used in the simulation are similar to those described in [18]. Typically, the construction and constraining of detailed simulation models requires a lot of experience and is a time-consuming task. If the corresponding simulation parameters and constraints are available, application-specific configuration scripts (e.g., material property or boundary condition settings for COMSOL) can be generated easily by an additional transformation. The methodology presented in this paper drastically reduces the simulation preparation time by using the XML-based ADF in combination with XSL transformations.

4 Design flow integration

This section brings together the different steps of the ADF methodology explained so far (i.e., input data processing, package design representation, model modifications, output data preparation, and interfacing with third party tools). The flowchart in **Figure 6** illustrates the flow. We used a scripting language (in our case Python) to "glue" the single steps into a flow, which is a common approach.

At the beginning, manual or tool-generated design data is transfered into an ADF representation. Three different paths are possible: design data is entered or exported directly in ADF, a suitable XSLT converts the input data from an intermediate XML to ADF, or an additional preceding step processes non-XML input data to XML first. Next, bond wires are inserted. The consideration of different accuracy levels (as described in **Section 3.2**) can be achieved in multiple ways (e.g., by passing variables into the XSLT processor, additional configuration files in XML, or corresponding configuration attributes or elements inside the input XML). In our example, a variable within the flow script is used to decide which sub-transformation is applied. Before the simulation is conducted, the STEP files are generated as described in the previous section.

The shaded process steps in Figure 6 are based on ADF. The internal structure is depicted in the flow chart comment box on the right side. If carefully designed (e.g., with clear and robust interfaces), many steps can be combined to form a problem-specific flow. The implementation of ADF using XML/XSLT technology is appealing due to its high flexibility.

ISBN 978-3-8007-4444-2

© VDE VERLAG GMBH · Berlin · Offenbach

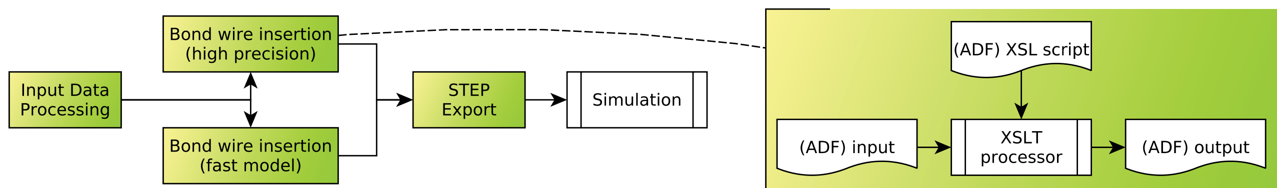

Figure 6 Example design flow to export a package model to simulation.

The presented flow enables automated simulation runs where a designer can quickly configure and investigate different package and assembly variants. A library of transformations with clearly defined interfaces can be used to describe and concatenate single design steps on an abstract level, such as "import from component list", "add wirebonds", and "export to STEP". To create or extend existing (packaging) design flows, only an XSLT processor is required (with platform independent and open source solutions available).

5 Summary and outlook

In this paper we presented a methodology to support the design of heterogeneous systems on package-level. The central aspect is the XML-based Assembly Description Format (ADF) and its modification using XSL transformations. ADF is able to consider geometries, electrical connectivity, material properties, simulation constraints, packaging rules, and further data related to package-level design. Furthermore, design steps and assembly processes can be modeled as well. Thus, ADF addresses requirements of a central design data base as well as of an underlying platform to implement an Assembly Design Kit (ADK) [7]. The applied software technologies are widely used and mature. This makes our approach also suitable for cloud-based design environments.

A stack of two wire-bonded bare dies in a LCC package served as design example. After a short introduction to ADF, we described several aspects of the design process (i.e., input processing, model transformations, data export). The integration with a simulation environment is only one possible application scenario.

The presented methodology can also be used to enable a sophisticated verification on package-level as described in [19]. Another important aspect in package-design is the export of manufacturing data [7]. Additionally, economical issues, such as supply chains and related assembly costs, can be incorporated. ADF enables the development of tools to explore and handle the big variety in packaging technologies. A long term goal is to allow the description of components not only in the electronic, but also in the mechanical, optical, and biochemical domain.

Acknowledgment

This work is supported by the BMBF within the project SiPoB3D. It was partially funded by the ECSEL Joint Undertaking under grant agreement No 737465 (named MICROPRINCE). The Joint Undertaking receives support from the European Union's Horizon 2020 research and innovation program and Germany, Belgium, Ireland.

References

[1] R. Fischbach, J. Lienig, and T. Meister, "From 3D circuit technologies and data structures to interconnect prediction," in *Proc. 11th Int'l Workshop on Syst. Level Interconnect Prediction*, SLIP '09, (New York, NY, USA), pp. 77–84, ACM, 2009.

[2] A. K. Varma, A. Glaser, and P. D. Franzon, "CAD flows for chip-package coverification," *IEEE Transactions on Advanced Packaging*, vol. 28, pp. 96–101, Feb. 2005.

[3] T. Brandtner, "Chip-package codesign flow for mixed-signal SiP designs," *IEEE Design Test of Computers*, vol. 23, pp. 196–202, May 2006.

[4] G. Brist and J. Park, "A novel approach to IC, package and board co-optimization," in *Sixteenth International Symposium on Quality Electronic Design*, pp. 512–518, March 2015.

[5] R. Ravichandran, J. Minz, M. Pathak, S. Easwar, and S. K. Lim, "Physical layout automation for system-on-packages," in *Proc. 54th Electr. Components and Technology Conference*, vol. 1, pp. 41–48 Vol.1, June 2004.

[6] J. M. Minz, E. Wong, M. Pathak, and S. K. Lim, "Placement and routing for 3-d system-on-package designs," *Trans. on Components and Packaging Technologies*, vol. 29, pp. 644–657, Sept. 2006.

[7] A. Heinig and R. Fischbach, "Enabling automatic system design optimization through assembly design kits," in *2015 International 3D Systems Integration Conference (3DIC)*, pp. TS8.31.1–TS8.31.5, Aug. 2015.

[8] Apache Xalan-Java. `https://xml.apache.org/xalan-j/`.

[9] Open Cascade. `https://www.opencascade.com/`.

[10] COMSOL. `https://www.comsol.com`.

[11] S. Wolf, A. Heinig, and U. Knöchel, "XML-based hierarchical description of 3D systems and SIP," *IEEE Design Test*, vol. 30, pp. 59–69, June 2013.

[12] W3C, "Extensible Markup Language (XML)." `https://www.w3.org/XML/`.

[13] W3C, "The Extensible Stylesheet Language Family (XSL)." `https://www.w3.org/Style/XSL/`.

[14] S. Ghali, *Introduction to Geometric Computing*, ch. Constructive Solid Geometry, pp. 277–283. London: Springer London, 2008.

[15] R. Fischbach, M. Dittrich, and A. Heinig, "Effizienter Design Rule Check von 3D Systemaufbauten mit einer hierarchischen XML-basierten Modellierungssprache," in *Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen, MBMV, Germany*, pp. 183–192, 2014.

[16] EIA/JEDEC, "Bond wire modeling standard." `http://www.jedec.org/sites/default/files/docs/jesd59.pdf`, June 1997. EIA/JESD59.

[17] A. Heinig, R. Fischbach, and M. Dittrich, "Thermal analysis and optimization of 2.5D and 3D integrated systems with Wide I/O memory," in *14th Intersociety Conf. on Thermal and Thermomechanical Phenomena in Electronic Systems (ITherm)*, pp. 86–91, May 2014.

[18] A. Heinig, D. Papaioannou, and R. Fischbach, "Model abstraction of 3D-integrated/interposer-based high performance systems for faster (thermal) simulation," in *15th Intersociety Conf. on Thermal and Thermomechanical Phenomena in Electronic Systems (ITherm)*, pp. 230–237, May 2016.

[19] R. Fischbach, A. Heinig, and P. Schneider, "Design rule check and layout versus schematic for 3D integration and advanced packaging," in *International 3D Systems Integration Conference (3DIC)*, pp. 1–7, Dec. 2014.