Change Management for Hardware Designers
From Natural Language to Hardware Designs

Martin Ring[1] Jannis Stoppe[1] Christoph Lüth[1,2] Rolf Drechsler[1,2]
[1]Cyber-Physical Systems, DFKI GmbH, 28359 Bremen, Germany
[2]Group of Computer Architecture, University of Bremen, 28359 Bremen, Germany

***Abstract*—To cope with rising hardware complexity, design processes are increasingly moved to more abstract description languages. Different descriptions impede the design process because they are usually disconnected. Therefore, adding more layers to the design process adds additional overhead to e.g. ensure that changes that are applied on the system level description are either done in accordance with other less or more abstract descriptions or that these changes are propagated accordingly. Managing these changes has so far been a manual task.**

This paper presents the *Change Impact Analysis and Control Tool* (CHIMPANC), a tool that uses state of the art analysis methods on various abstraction levels to build a single, interconnected model of these descriptions. These are used to track and manage any changes on each level of abstraction and their various refinement steps to ensure consistency throughout the development process.

The result is a tool that assists the developer by highlighting inconsistencies and required proof obligations across various descriptions in order to simplify the development process over various abstraction levels.

I. INTRODUCTION

The increasing complexity of hardware has long become the core issue of the underlying design and development processes. While systems with hundreds of millions of components can be manufactured, they need to be designed in the first place. Traditional hardware design languages (HDLs) are increasingly unable to handle designs this large. Traditional HDLs such as Verilog or VHDL describe systems that are supposed to be synthesised into hardware. They usually require designers to specify systems down to a point where they can be synthesised automatically. This entails that these designs need to be built from the bottom up and can only thoroughly be tested once the design is complete. Unfortunately, this approach cannot cope with requirements such as shorter design cycles and a reduced time to market.

One approach to remedy this issue is to provide designers with more abstract languages that allow systems to be designed top-down, starting with an abstract model of the system and its requirements. Several of these languages are being used today. Natural language specifications are the most abstract form of describing a system, allowing the designers to use arbitrary language to explain how the system is supposed to behave and be structured. Formal modelling languages such as the UML are built upon a formal definition to avoid the issue of ambiguities in the description. System-level modelling language such as SystemC are the last step before synthesizer HDLs, allowing to build virtual prototypes that can be simulated without actually implementing in the final hardware design.

Not only do these steps offer more abstraction than the traditional HDLs, they also form a natural hierarchy, from basically unrestricted natural language specifications, over more formalised ways to model a system down to an executable model without specifying its implementation details. These steps are supposed to be used subsequently: providing a natural language description first, then formalising it, providing a system level model and finally implementing the design at the register transfer level gradually leads designers through the process. However, when following this approach, several new challenges arise: firstly, we have to keep the models in the different levels of abstraction *consistent* across the different languages and formalisms involved, secondly, we need a uniform notion of *refinement*, and thirdly, we want to be able to *track changes* and their impacts across the different levels of abstraction.

This paper proposes the *Change Impact Analysis and Control Tool* (CHIMPANC) tool to handle these challenges. CHIMPANC extracts the relevant information from the models on the different levels and constructs mappings between them, thus allowing to check consistency and refinements, and moreover calculating the impact of changes. Thus, CHIMPANC ensures that *e.g.* a written specification or documentation is not made obsolete by changes in the implementation without being warned about it.

This paper will describe CHIMPANC, by first giving an overview of the different abstraction levels in Sect. II, outlining the various steps it performs to properly map levels and locate changes and check for consistency in Sect. III, and finally giving an overview over the front-end in Sect. IV.

II. ABSTRACTION LAYERS IN HARDWARE DESIGN

This section gives a short overview over different abstraction levels in system design, starting with the most abstract description and successively approaching traditional HDLs. The idea is that the ability to describe a system in a more abstract way means that details can be omitted early in the design process while retaining the ability to analyse the properties that have already been described.

A. Natural Language

The most abstract way to describe a system is natural language. When designing a system, specifying its properties without having to worry about details of mathematical notation, instead simply using the language one is familiar with is a straightforward way to start the design process.

Natural language basically does not restrict the designer in any way. It offers a way to describe a system with whatever words the author deems appropriate. This openness also means that this description cannot be formalised: while natural languages come with grammars that restrict the available constructs, these rules do not mean that the result is an unambiguous description of a certain system. While natural language processing techniques can address some issues, an automatic formalisation of arbitrary text is neither possible nor desired, meaning that these specifications need to be processed manually.

B. The Formal Specification Level (FSL)

The next step to describe a system in a more exact way are formal languages. Standardised languages such as the Unified Modeling Language (UML) give designers a way to describe the system readily but at the same time force them to adhere to a formal grammar that makes these descriptions more or less unambiguous [1]. UML thus offers a way to add precision to the system description.

Still, this formalised notation does not specify all aspects of the system; *e.g.* the UML lacks the ability to express non-functional requirements such as timing properties. In other words, FSL models are merely constraining the properties of the design, *e.g.* by structural diagrams enriched with OCL constraints which limit what actions may be performed by the system and how the output values may then be structured. However, while these models may be used to locate potential errors early on in the design process, they are neither complete nor actually executable.

C. The Electronic System Level (ESL)

The next step in refining the system is to create a working prototype without going into the implementation details required by HDLs. System level modelling languages such as SystemC can describe the behaviour of systems without specifying how this functionality is supposed to be implemented.

SystemC, as the current de-facto ESL standard language [2], allows systems to be described using the C++ programming language while at the same time offering designers the means to describe the structural features of a hardware design. The result is a virtual prototype that can be simulated: parts that are meant to represent hardware are managed by a dedicated simulation kernel which invokes the relevant software parts.

This means that the ESL design is much less abstract than at the FSL, representing a model of the system that can be executed, while still being too abstract to be translated into hardware.

D. The Register Transfer Level (RTL) and below

The Register Transfer Level gives designers the ability to design systems that may be translated into hardware [3]. Dedicated HDLs are specifically designed to be mapped to hardware, focusing on the description of structural features and parallel execution while at the same time limiting the designer concerning elements that cannot be built as hardware parts such as loops (which need to be unrolled and hence bounded). Where ESL models can just specify that a module calculates a result using arbitrary means (such as a call to a given software library), RTL designs need to specify how exactly the results are computed.

E. Different Levels of Abstraction

These different abstraction levels all have particular purposes and use cases:

- natural language offers a way to quickly come up with an initial description of a given system that is well-readable without prior training and not restricted concerning the described properties;
- FSL models specify system properties in a precise way amenable to formal analysis and reasoning;
- ESL models offer virtual prototypes to be run and tested;
- RTL implementations allow the design to be translated into hardware.

All the different levels describe the same system, yet they are written in different and at first sight unconnected languages. Thus, we need to ensure that the models at the different abstraction levels are consistent: the natural language requirements need to be represented as formal properties at the FSL, the classes modelled at the FSL need to appear in the ESL as well, *etc.* Further, one abstraction level may contain several models of the system at different degrees of abstraction: at first, an FSL model should be no more than a translation of the natural language requirements, while a more detailed FSL model should be detailed enough such that we can translate it into the ESL and SystemC. This is called *refinement*: gradually adding more details which constrain the model of the system. Keeping the models occuring throughout the development consistent with each other is called functional change management.

III. Functional Change Management

Functional change management calculates the impact of syntactical changes using the semantics of the documents. In order to implement functional change management across the different levels of abstraction, we need a unifying semantics for the different levels.

A. Underlying Semantics

In our case, the semantics is based on Kripke structures. Without going into the details here, a Kripke structure consists of a set of states, a transition relation between states, and a set of propositions which hold at each state. Thus, Kripke structures allow us to capture the key notions of state transition and state-dependent predicates.

We now sketch the semantics of each of our levels. The NLP cannot have a mathematically precise semantics, as such would counteract our motivation to use natural language in the first place (we want users to be able to express initial specifications without having to worry about mathematical rigour at the same time). Instead, we use decompose the natural language requirements into single semantically meaningful requirements, which are subsequently mapped to the lower levels.

In the FSL, the class and object diagrams give us a notion of state (see [4] for details): classes describe the system state (via an object model), and object diagrams describe particular system states (in particular, initial states). State transitions are given by the OCL constraints: there is a transition with operation o from S_1 to S_2 iff. (i)

1) all invariants hold in S_1 and S_2,
2) the preconditions of o are satisfied in S_1, and
3) the postconditions of o are satisfied in S_2.

Additionally, transitions can be specified using state diagrams. Thus, the semantic entities here are classes, invariants, pre- and postconditions, objects, or state diagrams.

In the ESL, the semantics are given by the SystemC semantics. States are given by the instances of the SystemC modelling classes (`sc_module` etc.), and transitions are given by the simulation (see [5] for details; however, we use a reasonable abstraction from the concrete SystemC implementation instead of a mathematically precise model of the implementation). Thus, our semantic entities here are classes, attributes, and methods.

The semantic entities on the respective abstraction levels give rise to notions of mapping between them. From the natural language level to FSL and ESL, we map each requirement to one or more specification elements which implement them. Within the FSL, we define a notion of refinement; essentially, a concrete specification $\mathcal{C}$ is a refinement of an abstract specification $\mathcal{A}$ if all state transitions in $\mathcal{C}$ (corresponding to a trace in the underlying Kripke structure) can be mapped back to a state transition in $\mathcal{A}$, i.e. $\mathcal{C}$ restricts the possible state transitions of $\mathcal{A}$. This refinement can be more concretely realised by refining the state (data refinement) or the operations (operational refinement), or combinations thereof. An example of the former is the introduction of new classes or attributes, an example of the latter is the implementation of a single operation by a state diagram. From the FSL to the ESL, we have the usual implementation of UML diagrams, except that we may map classes in the FSL to instances of the `sc_module` class (corresponding to the fact that in hardware, objects exist more or less *a priori*). Within the ESL (e.g. between two SystemC models), we do not consider refinement.

A system development consists of several *layers* $L_1, \ldots, L_n$, each of which contains one or more specifications from one of the abstraction levels described above. The first layer typically contains the natural language specifications, and the last layer L_n ESL or RTL specifications. Between the layers, specifications are related via refinement: a specification SP from layer L_i is mapped to a specification SP' of layer L_{i+1} if SP' is a semantic refinement. This mapping allows us to keep track of properties; for example, if all initial NL requirements are mapped to formal properties which are later proven we can be confident that the implementation satisfies the original specifications.

The mappings are mostly constructed automatically (see Sect. III-E below), but partly have to be constructed by the user (in particular, the one mapping the NL requirements).

B. A Common Data Structure

The specifications on the different levels are written in a variety of different formalisms, each in their own syntax. Since we aim to extensibly support a wide variety of file types reaching from natural language to low level hardware descriptions, it would be inflexible to implement a direct adaptor for every input syntax. Hence we decided to employ the widely adopted, generic Eclipse Modelling Framework (EMF) [6], which then serves as a common basis for other file types. This means that any format is supported as soon as there is a translation into EMF. For the FSL this does not incur any additional overhead if we use the UML tools provided by the EMF. Natural Language is currently simply represented as a list of SysML requirements. For SystemC, we use the debug output of the clang compiler to generate an EMF model, including namespace and class structures with type hierarchies, operations and attributes. RTL formats like VHDL or Verilog can be supported by providing a transformation into EMF but that is not yet implemented.

C. Syntactic Difference Analysis

The underlying architecture of the functional change management has been derived from the GMoC system [7]. A generic diff algorithm for hierarchical annotated data serves as a basis [8], and provides support for syntactic difference analysis. We adapted this algorithm to operate on generic EMF objects (EObjects). This way we can obtain a minimal set of changes between two EMF files. The GMoC diff algorithm allows us to specify equivalence between the objects; in our case, which attributes identify an object, which orderings have a meaning and which do not. The example in Fig. 1 states that a UML class is identified by its name, and that the order of the contained attributes and operations is irrelevant, while on the other hand the order of the parameters of an operation has a semantic meaning.

```
element EClass {
  annotations {
    name!
  }
  constituents {
    unordered { _ }
  }
}
```

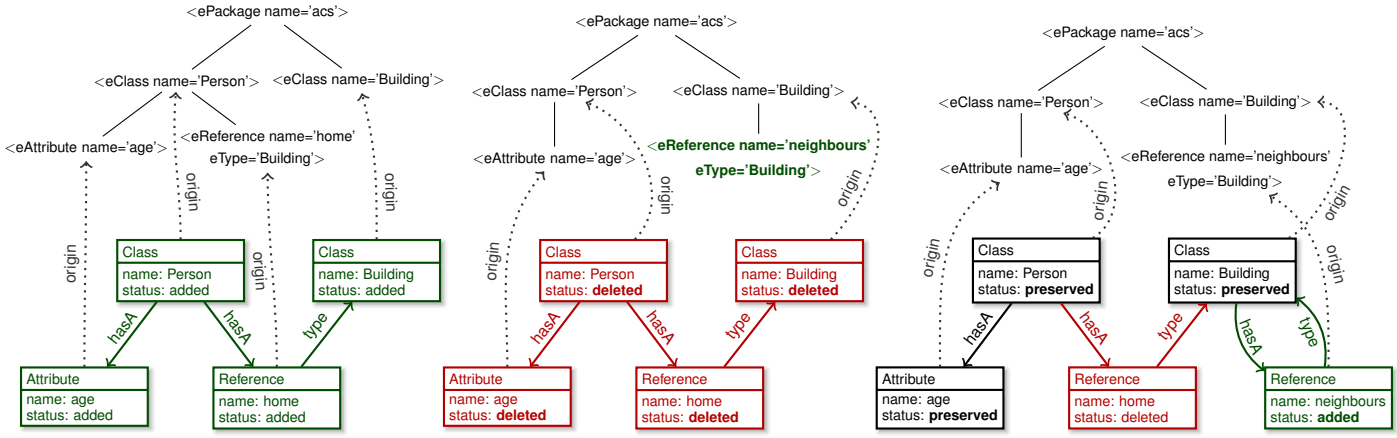
```
element EOperation {
  annotations {
    name!
  }
  constituents {
    ordered { _ }
  }
}
```

Fig. 1. Example `ecore.equivspec` file

We chose the industry-proven Neo4j graph database for persistence, because it allows us to efficiently traverse and transform the abstract syntax tree while providing superb scalability. On top of this we implemented an interface from EMF to Neo4j which allows us to analyse differences between files on disk and the persisted syntactic tree in the database.

D. Semantic Difference Analysis

The distinctive feature of the diff algorithm is that it takes the intended semantics of the documents into account. This is achieved by representing the semantics by a graph as well(*explicit semantics*). The semantic graph is extracted from the syntactic graph by graph rewrite rules, which can be



(a) after initial extraction (b) after application of syntactic diff (c) after second extraction

Fig. 2. Change management via explicit semantics

efficiently implemented in Neo4j; after extraction, the nodes of this semantic graph are connected to the origin nodes of the syntactic tree (Fig. 2(a)).

When a change in an input file occurs, a diff is applied to the syntactic tree. Then, we mark the nodes of the semantic graph as "deleted" (Fig. 2(b)) and extract the graph again (Fig. 2(c)). Nodes that are already present in the graph are marked as "preserved", nodes that do not exist are marked as "added", and all other nodes remain marked as "deleted". During this process additional semantic knowledge can be used to handle individual nodes as required.

E. Change Propagation Across the Layers

The semantic graphs of specifications from adjacent layers can be mapped semi-automatically by inspecting naming, types and structure of models. Users are always in control of these mappings and can alter or complement them where required to reflect their intentions.

Change propagation follows syntactic changes across the origins along the mappings of the semantic graph. That is, if a syntactic change occurs we find which parts of the semantic graph have their origins in the that part of the syntactic graph which has changed, and then check which mappings either point into, or originate from this part of the semantic graph. For example, suppose we have three layers L_1, L_2, L_3 with classes C_1 in L_1 implemented by C_2 in L_2 and class C_2 in L_2 implemented by class C_3 in L_3. Classes C_1, C_2 and C_3 contain references R_1, R_2 and R_3 respectively, where R_2 implements R_1 and R_3 implements R_2. The user might change the type of R_2. This change affects the abstraction R_1 and the refinement R_3 and might lead to inconsistencies on either side. If an operation is inserted into C_2 this affects only the refinement C_3 since an operation does not have to be present in the abstraction C_1 for C_2 to be a valid refinement. A more complex example arises if we look at proof obligations that arise from refined OCL constraints. These proof obligations are of the form $c_1 \wedge \ldots \wedge c_n \implies d$, where c_1 to c_n are constraints on the refined level and d is a constraint in the abstract level. If we prove this externally (our tool does no OCL reasoning), we can discharge the obligation and insert additional dependency edges between the constraints $c_1 \ldots, c_n$ and d. If one of these constraints changes the proof will be invalidated and the proof obligation pops up again. Impact rules such as these are described directly as CYPHER queries; this makes the impact system extensible.

IV. The ChimpanC Tool

In the previous section it became apparent that functional change management needs user interaction, and hence an intuitive and visual user interface. Existing tools that encompass this workflow are rare and usually focus on a single, specific aspect such as natural language processing [9] or SystemC code generation [10] – a sophisticated cross-layer change management tool has yet to be developed. ChimpanC is our answer to this. It implements the basic concepts of Sect. III, and allows the user to easily inspect, modify and augment the refinement mappings. On top of this we visualise how changes in one layer affect the other layers.

A. Proposed Workflow

We envision a design workflow which is compatible with existing hardware design process models established in the industry. Since in practice there exists a heterogeneous tool infrastructure ranging from word processors down to specialised tools for circuit design, all used by a diversity of people with different levels of understanding, it would be impractical to integrate our change management into all of these tools. Hence, we propose the ChimpanC tool as an independent augmentation of the process. While the different designers keep using their accustomed tools they get the possibility to define and inspect relations between formerly unrelated layers and gain a new, richer perspective on the design. This process becomes even more valuable thanks to the automatic mappings which our tool constructs.

To relate refinement layers the users start by defining a project definition file. Here they declare the refinement structure by providing a list of layers, each with a type indicator (currently one of the values `isl`, `fsl`, `esl`) for the respective specification level and an arbitrary list of files belonging to this layer. After this initial project definition is done, the tool can be used.

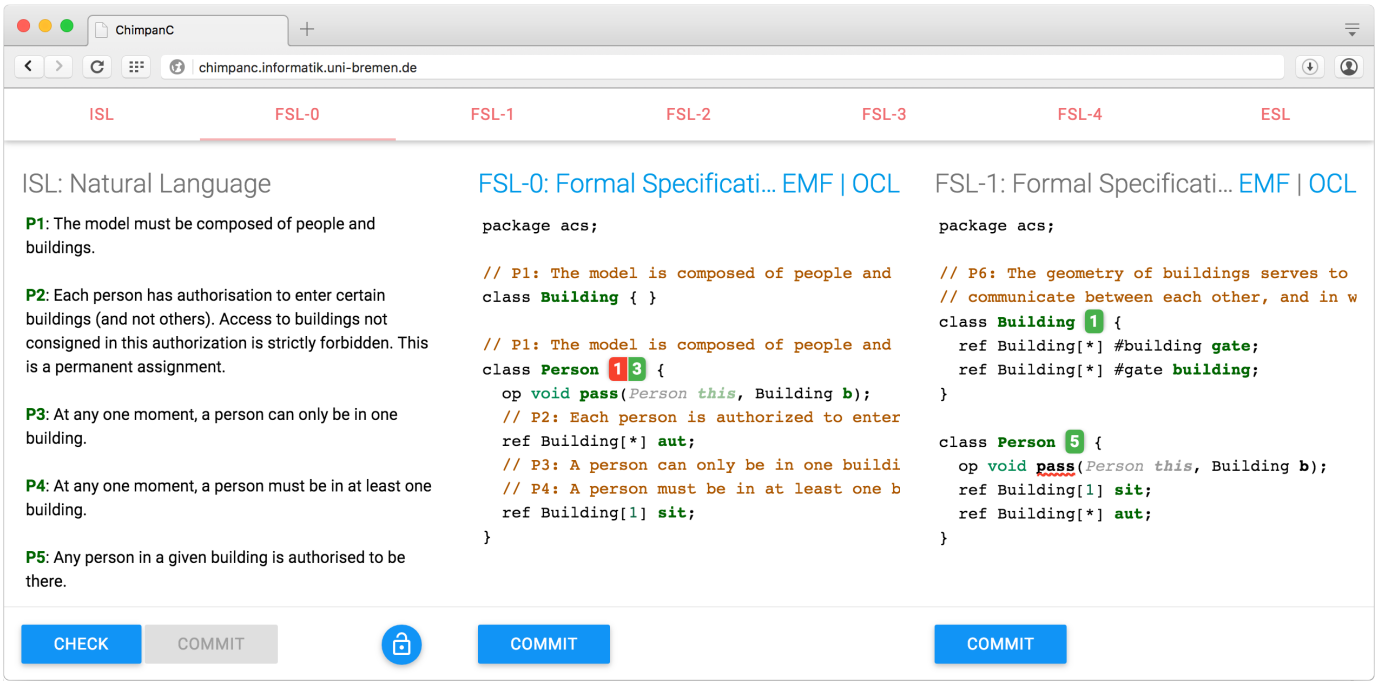


Fig. 3. The CHIMPANC user interface.

B. The User Interface

CHIMPANC is realised as a web interface and can thus either run locally or on a team server, configured for a specific system that is being developed. When users open the application in a browser they get presented a multi column layout representing the different specification layers (Fig. 3). The leftmost column is the most abstract one — typically natural language — while every additional column to the right represents a refinement step. There are usually more refinement steps involved than would fit into the user interface, so there is a navigation bar on the top where one can select the layer in focus.

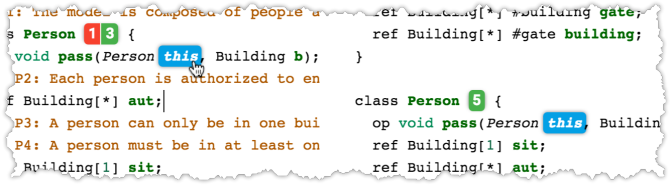


Fig. 4. Highlighting of mappings

All extracted model elements are represented as bold identifiers. Mapped model elements appear green. When a user hovers the mouse over such a mapped element the corresponding refinement is visually emphasised (Fig. 4).

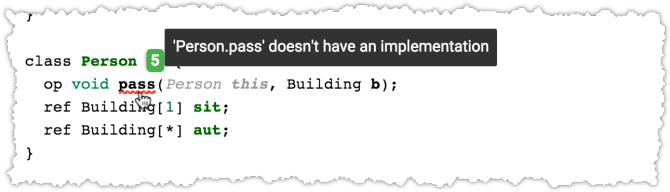


Fig. 5. Highlighting of inconsistencies

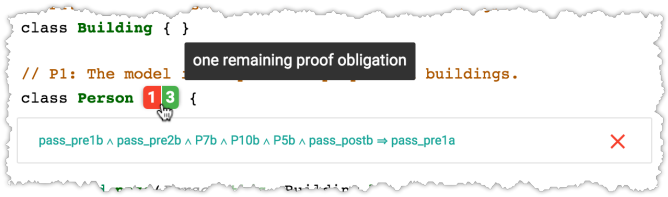


Fig. 6. Inline display of proof obligations

Inconsistencies are highlighted with red squiggly underlines. These include abstract models, attributes, references, operations and parameters which are unmapped in a refinement (Fig. 5) as well as mismatching mapped types and inconsistent multiplicities of references. In addition, unproven OCL refinements are displayed as a red number next to the respective class definition which indicates the number of open proof obligations on the other hand discharged proof obligations appear as a green number (Fig. 6). When the user moves the mouse over a marked element a tooltip will appear, containing information about the inconsistency.

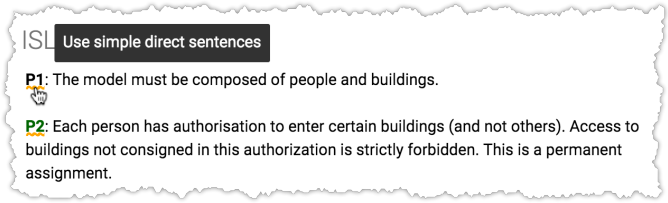


Fig. 7. A content warning in natural language

Content warnings are highlighted with orange squiggly underlines. These are currently only present in natural language where we automatically rate the quality of refinements, using

the techniques from [11]. Again, a detailed description of the warning can be obtained by hovering the mouse over the marked element (Fig. 7).

Fig. 8. An impact warning

Finally impact warnings appear as orange elements indicating that user attention is required (Fig. 8). An impact can either indicate that a refinement has changed or that the abstraction has been changed or removed. Impacts warnings are the default fallback when there is no automatic solution to propagate a change across layers. It still offers a high value to developers because the possibly affected portions of refinements and abstractions can are narrowed down to small fractions of the specification and inconsistencies can easily be identified. Removed refinements do not trigger an impact warning because they already result in an inconsistent model, and thus an inconsistency error.

V. Conclusion

We presented ChimpanC, a tool which supports a comprehensive system design flow across different levels of abstraction levels, from natural language down to system-level models. ChimpanC manages the models of the systems at the different abstraction levels, keeps track of dependencies, and calculates the impact of changes. Furthermore, it can warn about inter layer inconsistencies that would previously be left unnoticed by the established tool chain.

We believe that our tool is easy to integrate into existing workflows since it is independent of the utilised tools and can be extended to support all kinds of formats using EMF as a simple and well documented interface. Users can provide rules for refinement and automatic impact propagation as graph rewriting rules in the form of Cypher queries. Even if not all designers in a team use the tool, it offers added value, since it provides a way to detect and communicate the impact of changes across different layers.

A. Related Work

There are several independent approaches to change management for some of the individual specification levels we described. EMF itself for example offers a toolset to analyse differences between two models [12] and there are entire change management systems for UML [13]. However, these systems share several limitations, the foremost being that there are no semantic connections to external models taken into consideration, leaving the user without knowledge about impacts to other specification layers. Also we are not aware of any other change management tool available which is able to calculate the impact of changes on the correctness of UML/OCL refinements. In addition, ChimpanC supports impact analysis between SystemC and the FSL as well as between natural lanugage and the FSL.

B. Future Work

The integration of RTL as well as formal semantics for SystemC refinements are still required to depict the entire hardware design workflow.

The mapping from natural language to FSL is currently manual. We are evaluating NLP techniques to partly automate this process. Automatic change propagation rules from FSL to natural language would be very valuable since they would imply drastically enhanced means of communication between designers and stakeholders and remove a lot of possibility for misunderstandings.

To move the tool out of the prototype status, we are planing to conduct a large case study together with industry partners.

VI. Acknowledgements

The research reported here was supported by BMBF under grant 01IW13001 (SPECifIC).

References

[1] R. Drechsler, M. Soeken, and R. Wille, "Formal specification level," in *Models, Methods, and Tools for Complex Chip Design*, ser. Lecture Notes in Electrical Engineering, J. Haase, Ed. Springer International Publishing, 2014, vol. 265, pp. 37–52.

[2] C. Schulz-Key, M. Winterholer, T. Schweizer, T. Kuhn, and W. Rosentiel, "Object-Oriented Modeling and Synthesis of SystemC Specifications," in *Asia and South Pacific Design Automation Conference (ASP-DAC)*. IEEE, 2004, pp. 238–243.

[3] L. J. Hafer and A. C. Parker, "A formal method for the specification, analysis, and design of register-transfer level digital logic," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 2, no. 1, pp. 4–18, 1983.

[4] M. Richters and M. Gogolla, "OCL: Syntax, Semantics, and Tools," in *Object Modeling with the OCL*, ser. Lecture Notes in Computer Science, T. Clark and J. Warmer, Eds. Springer Berlin Heidelberg, 2002, no. 2263, pp. 42–68.

[5] *Standard SystemC Language Reference Manual*, IEEE, iEEE Standard 1666 – 2011.

[6] D. Steinberg, F. Budinsky, E. Merks, and M. Paternostro, *EMF: eclipse modeling framework*. Pearson Education, 2008.

[7] S. Autexier and N. Müller, "Semantics-based change impact analysis for heterogeneous collections of documents," in *Proceedings of 10th ACM Symposium on Document Engineering (DocEng2010)*, M. Gormish and R. Ingold, Eds., Manchester, UK, september 2010.

[8] S. Autexier, "Similarity-based diff, three-way-diff and merge," *International Journal of Software and Informatics (IJSI)*, vol. 9, no. 2, august 2015.

[9] O. Keszocze, M. Soeken, E. Kuksa, and R. Drechsler, "Lips: An ide for model driven engineering based on natural language processing," in *Natural Language Analysis in Software Engineering (NaturaLiSE), 2013 1st International Workshop on*. IEEE, 2013, pp. 31–38.

[10] V. Sinha, F. Doucet, C. Siska, R. Gupta, S. Liao, and A. Ghosh, "Yaml: a tool for hardware design visualization and capture," in *Proceedings of the 13th international symposium on System synthesis*. IEEE Computer Society, 2000, pp. 9–14.

[11] M. Soeken, N. Abdessaied, A. Allahyari-Abhari, A. Buzo, L. Musat, G. Pelz, and R. Drechsler, "Quality assessment for requirements based on natural language processing," in *Forum on Specification and Design Languages - Proceedings*. o.A., 2014.

[12] The Eclipse Foundation. (2012) EMF Diff/Merge. [Online]. Available: http://www.eclipse.org/diffmerge/

[13] L. C. Briand, Y. Labiche, and L. Sullivan, "Impact analysis and change management of uml models," in *Software Maintenance, 2003. ICSM 2003. Proceedings. International Conference on*. IEEE, 2003, pp. 256–265.