Change Impact Analysis for Hardware Designs
From Natural Language to System Level

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

***Abstract*—Design processes are increasingly moving to more abstract description levels; no single formalism can handle the complexities of modern designs. However, keeping designs consistent across different abstraction levels, in particular in the presence of changes, has up to now been an arduous manual task.**

This paper presents a framework which provides a uniform, interconnected representation of the descriptions across the abstraction levels, starting from natural language requirement specifications over SysML design specifications down to executable SystemC models, allowing to track changes on all levels of abstraction, and ensuring consistency throughout the development process.

The framework has been implemented in a tool, CHIMPANC, to show its viability. It assists the developer by highlighting inconsistencies and proof obligations across various descriptions levels in order to simplify the development process.

I. INTRODUCTION

The increasing complexity of hardware has long become the core issue of the underlying design and development processes. Traditional hardware design languages (HDLs) such as Verilog or VHDL which are supposed to be synthesised into hardware are increasingly unable to handle large designs, because they require designers to specify systems to the point where they can be synthesised automatically. The resulting designs need to be built from the bottom up and can only be verified by thorough testing once complete. This approach cannot cope with the shorter design cycles and reduced time to required in today's marketplace.

The remedy suggested in this paper 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 or SysML build on a formal definition to avoid the issue of ambiguities in the description. System-level modelling language such as SystemC are the last step before synthesizable HDLs, allowing to build virtual prototypes that can be simulated without actually implementing in the final hardware design.

These languages form a hierarchy, and 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.

The contribution of this paper is a framework which aims at meeting these challenges. The framework provides a uniform management of specifications in these languages at a syntactic level, a semantics to relate their meaning (as far as possible) by a notion of refinement, and a comprehensive change management across all levels. We have implemented the framework in a prototype of the *Change Impact Analysis and Control Tool* (CHIMPANC) to demonstrate its principal applicability. It is particularly the change management which makes this approach viable, because we need to be able to handle changing specifications effectively; changes are the norm, rather than the exception, as the design will rarely be correct the first time, and moreover the tool supported afforded at the more abstract levels will help us to find errors earlier in the design process, necessitating these changes.

This paper is structured as follows: we first give an overview of the different abstraction levels in Sect. II, then outline the various steps it performs to map levels, locate changes and check for consistency in Sect. III, and finally give an overview of the front end in Sect. IV.

II. HARDWARE DESIGN ABSTRACTIONS

This section gives a short overview over different abstraction levels in system design, starting with the most abstract description and successively approaching traditional HDLs.

A. The Informal Specification Level (ISL)

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 and simply using the language one is familiar with instead is a straightforward way to start the design process.

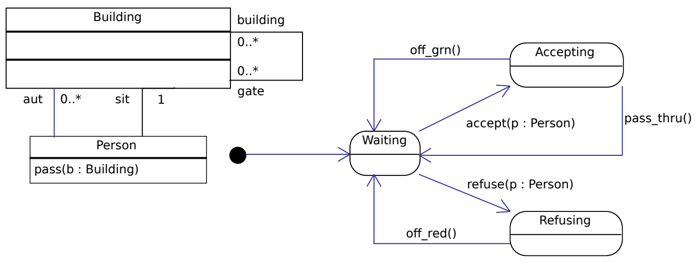
Natural language does not restrict the designer in any way. This openness means that this description cannot be formalised: while natural languages come with grammars that restrict the available constructs, these rules do not mean that the result is an unambiguous description of the system. While natural language processing (NLP) 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 Systems

P1: The model is composed of people and buildings.

P5: Any person in a given building is authorised to be there.



context Person
 inv P5: **self**.aut→includes(**self**.sit)

```
SC_MODULE( Door )
{
  ...
  LED grn;
  LED red;
  Turnstile ts;
  Gate gc;
}

SC_MODULE( Gate )
{
 ...
 void operate()
 {
 ...
 }
};
```

Informal Specification Level | Formal Specification Level | Electronic System Level

Fig. 1. Example Development of an Access Control System (excerpt): from Informal Specification Level to Electronic System Level.

Modeling Language (SysML) 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 unambiguous [1]. SysML 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 SysML lacks the ability to express non-functional requirements such as timing properties. In other words, FSL models formalise the constraints inherent in the design; *e.g.* structural diagrams enriched with OCL 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.

From the ESL, we can map the system design further down to dedicated HDLs which may be translated into hardware [3]; this is called the Register Transfer Level (RTL), but as the connections here are fairly well-known we will not explore it further in this paper.

D. 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 (not considered here) would 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 at the ESL in corresponding form *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 SystemC at the ESL. This is called *refinement*: gradually adding more details which constrain the model of the system. Keeping the models throughout the development consistent with each other is called *functional change management*.

E. Example: an Access Control System

As an example, consider the design of an access control system (appropriated from [4]). It should control the access of people to buildings by controlling the doors. Initial natural language requirements state facts about their relations (Fig. 1, on the left).

To formalise requirements such as these, we introduce SysML blocks, with added OCL constraints. Here, classes initially include people and buildings; associations include AUT and SIT, which point to the buildings someone is authorized to enter, or is currently in, respectively. There is one OCL constraint which states that every person can only be in a room she or he is authorized for, *i.e.* SIT $\in$ AUT, written in OCL as **self**.aut→includes(**self**.sit) (Fig. 1, middle-left). These formalized but very loose constraints can now be refined further. For example, we introduce doors which connect buildings, and people are authorized to access certain doors. To make this into an ESL specification, we then describe the actual mechanics of operating the door in more detail: when a person is approaching the door, a green or red light should indicate whether access is granted or denied, and a turnstile should open (or not). This can be expressed by a state machine diagram in SysML (Fig. 1, middle-right).

In our refinement steps, we have replaced modelling classes such as people and buildings by implementation classes like doors. The final refinement step translates a state machine diagram into a SystemC implementation, with doors (but not people) becoming components (called SC_MODULE in SystemC), comprised of a card reader, a turnstile, and green and red LEDs. The turnstile has a method operate which implements the state machine diagram above (Fig. 1, right).

III. A Framework for Change Impact Analysis

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

Note that the methods proposed in this paper and the tool introduced are indeed focusing on *change management*. The use case is *not* to generate a formal description of a system from a natural language specification, but to detect which parts of specifications are related to parts of other specifications, and how changes in one of them affect the other, *i.e.* how changes propagate within and across the different levels of abstraction.

A. Related Work

Change impact analysis offers more than the currently used source code management (SCM) tools (Git, Subversion, Mercurial, *etc.*); our work does not compete with any of these but augments them with functional change management, and the proposed solution could be easily integrated into any of these existing SCM solutions.

There are several isolated approaches to functional change management for some of the individual specification levels we described. EMF itself for example offers a toolset to analyse differences between two models [5], there exists a change management systems for UML diagrams [6], and there is a wealth of techniques on traceability and requirements management [7], [8]. However, these systems share several limitations, the foremost being that there are no semantic connections to external models which could be taken into consideration, leaving the user without knowledge about impacts to other specification layers. Furthermore, we are not aware of any other change management tool available which is able to calculate the impact of changes on the correctness of SysML/OCL refinements. In addition, ChImpAnC supports impact analysis between different abstraction levels.

The analysis of SystemC designs is a complex task that is a research field on its own. Embedding SystemC into a change managed workflow is thus a non-trivial task as a various C++ dialects need to be supported, each tied to compilers that generate an optimized binary version of the design to be run that is stripped of all non-essential meta information. Different approaches to extract the given information include parsing the source code [9]–[13] (which results in the support of only a subset of SystemC, as no existing parser supports all given dialects) or using modified compilation workflows in order to modify the executable design to trace and store the required data itself [14]–[16] (which results in the support of all designs that are built using the compiler being used). In order to keep our approach as applicable as possible, the approach given in [17] was used: instead of relying on the source code, the compiler-generated debug symbols are used. While the format itself differs between compiler architectures, it is always standardized and/or accessible, resulting in a reliable interface to retrieve structural descriptions from SystemC designs.

The OCL approach to specification with preconditions, postconditions and invariants is called design by contract and goes back to [18]. More recently, this approach can be found in component-based design (rich components [19]), or so-called light-weight specification languages based on a programming language, such as JML [20] for Java or ACSL [21] for C. The latter two focus on what corresponds to the lowest abstraction layer in our setting, the ESL. Existing tools for the whole workflow across all abstraction layers are rare; most closely related are so-called wide-spectrum languages [22] which cover the whole of the design flow. For example, our running example was originally conceived for the B language [23]. Atelier B, the tool supporting B, covers the whole design flow, similar to Event-B, an extension of B with events, which is supported by the Rodin tool chain [24]. Another prominent example is SCADE [25], which supports seamless and rigorous development from abstract specification down to executable software or RTL code by code generation techniques. The drawback of all these languages and tools is that they tie the user into one language and methodology for the whole design flow, whereas our approach offers designers a best-of-breed approach, and integrates into existing design flows. Moreover, we are not aware of any attempts to apply impact analysis on any of these wide-spectrum languages.

B. Underlying Semantics

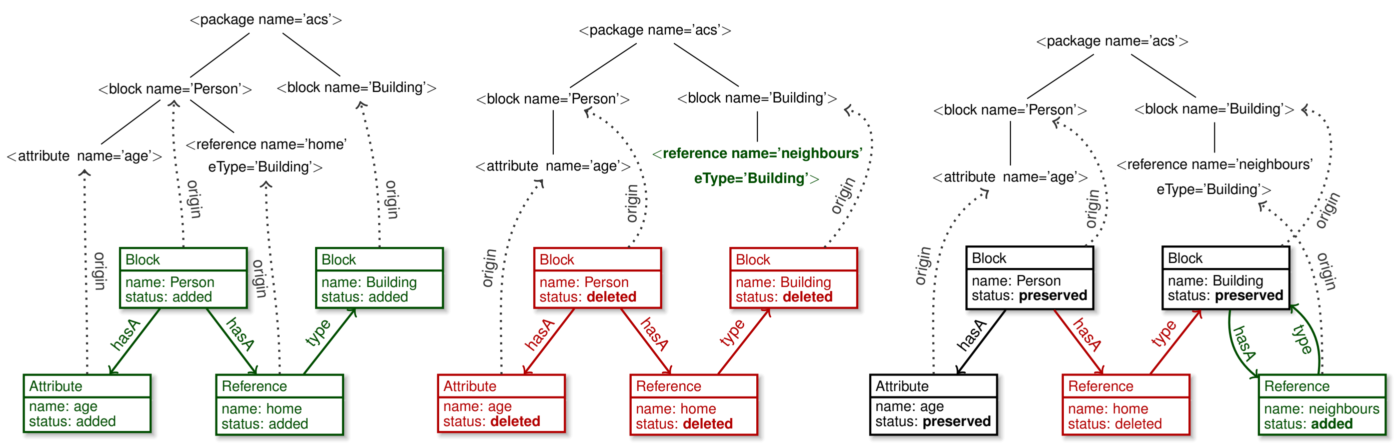
In our case, the semantics is based on Kripke structures. Without going into the details here, a Kripke structure consists of a set of states, a transition relation between states, and a set of propositions which hold at each state. Thus, Kripke structures allow us to capture the key notions of state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

We now sketch the semantics of our abstraction levels. The ISL 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 NLP techniques to decompose the natural language requirements into single semantically meaningful requirements, which form the semantic entities at the ISL. Additionally, if NLP does not offer satisfying results, connections between elements of the FSL and the ISL can be drawn manually in order to properly detect the impact of changes across the different abstraction levels.

At the FSL, the class and object diagrams give us a notion of state (see [26] 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) all invariants hold in S_1 and S_2,
(ii) the preconditions of o are satisfied in S_1, and
(iii) the postconditions of o are satisfied in S_2.

Additionally, transitions can be specified using a restricted form of state diagrams. Thus, the semantic entities at the FSL



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

Fig. 2. Change management via explicit semantics

are classes, invariants, pre- and postconditions, objects, and state diagrams.

At 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 [27] for details; however, we use a reasonable abstraction from the concrete SystemC implementation instead of a mathematically precise model of the implementation). Thus, the semantic entities at the ESL are classes, attributes, and methods.

The semantic entities on the respective abstraction levels give rise to notions of mapping between them. From the ISL 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 based on the underlying Kripke structures; a concrete specification $\mathcal{C}$ is a refinement of an abstract specification $\mathcal{A}$ if each state transition in $\mathcal{C}$ 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 realised by refining the state (data refinement) or the operations (operational refinement). An example of data refinement is the introduction of new classes or attributes; an example of operational refinement is the implementation of a single operation by a state diagram. From the FSL to the ESL, we have the usual implementation of SysML diagrams, except that we may map blocks 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 (*i.e.* between two SystemC models) we do not consider refinement, as this would require a more sophisticated semantic modelling of SystemC to consider *e.g.* timing requirements.

A system development consists of several *layers* $L_1, \ldots, L_n$, which group specifications from one abstraction level. The first layer typically contains the natural language specifications, and the last layer L_n ESL specifications. Between 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 ISL 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-F below), but some have to be constructed by the user (in particular, the mapping of ISL requirements).

C. Syntactic Representation

The specifications on the different levels are written in different formalisms, each in their own syntax. Since we aim to extensibly support a wide variety of file types in the future, it would be inflexible to implement a parser for every concrete input syntax. Hence we decided to employ the widely adopted, generic Eclipse Modelling Framework (EMF) [28], which 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. At the ISL, specifications are represented as a list of SysML requirements. At the FSL, we use the SysML tools provided by the Papyrus Framework [29], as well as the EMF OCL representation. For the ESL, we make use of the fact that SystemC models are valid C++ source files, and employ the debug output of the clang compiler to generate an EMF model. The files contain DWARF debug information that can be extracted using the libdwarf/dwarfdump tools. The resulting data is translated to the EMF format using a custom parser/translator. The final result includes namespace and class structures with type hierarchies, operations and attributes.

D. Syntactic Difference Analysis

The architecture of the functional change management has been derived from previous work in the generic GMoC system [30]. A generic diff algorithm for hierarchical annotated data serves as a basis [31], 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. 3 states that a SysML block 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 Block {                    element Operation {
  annotations {                      annotations {
    name!                              name!
  }                                  }
  constituents {                     constituents {
    unordered { _ }                    ordered { _ }
  }                                  }
}                                  }
```

Fig. 3. Example `ecore.equivspec` file

E. 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 as a graph (*explicit semantics*). The semantic graph is extracted from the syntactic graph by graph rewrite rules, which can be 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.

Thus, we have the *syntactic graph* which consists of the abstract syntax trees, and the *semantic graph* extracted from them. We store both graphs uniformly in the Neo4j graph database, because it allows us to efficiently traverse and transform them 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.

F. Change Impact Analysis

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 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. To illustrate, consider our example (Fig. 1): ISL requirement P5 (left) gets mapped to OCL invariant P5 (middle-left); if either the requirement or the OCL invariant is changed, the other is impacted by the change, as inconsistencies between the two might arise. If the user changes the state diagram in the ESL, this change might impact the ESL implementation, or on the other hand the OCL invariants of the class diagram.

For data or operation refinements, we can calculate the impact of changes more accurately. If we add additional operations to the class `Building` in Fig. 1, all data refinements of `Building` will remain valid. The situation gets more complex when we consider the proof obligations that arise from refined OCL constraints. These proof obligations are of the form $c_1 \wedge ... \wedge c_n \implies d$, where c_1 to c_n are constraints on the refined level and d is a constraint in the abstract level. If this is proven 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 Neo4j queries; this makes them fast to execute and keeps the impact system extensible.

IV. The ChImpAnC Tool

Automation notwithstanding 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 [32] or SystemC code generation [33]. In contrast, ChImpAnC is a sophisticated cross-layer change management tool. 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 it visualises 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, or to combine all of these tools into a new IDE. Hence, we propose the ChImpAnC tool as an independent augmentation of the process. Designers keep using their accustomed tools to edit specifications, but additionally get the possibility to define and inspect relations between the layers and gain a new, richer perspective on the design.

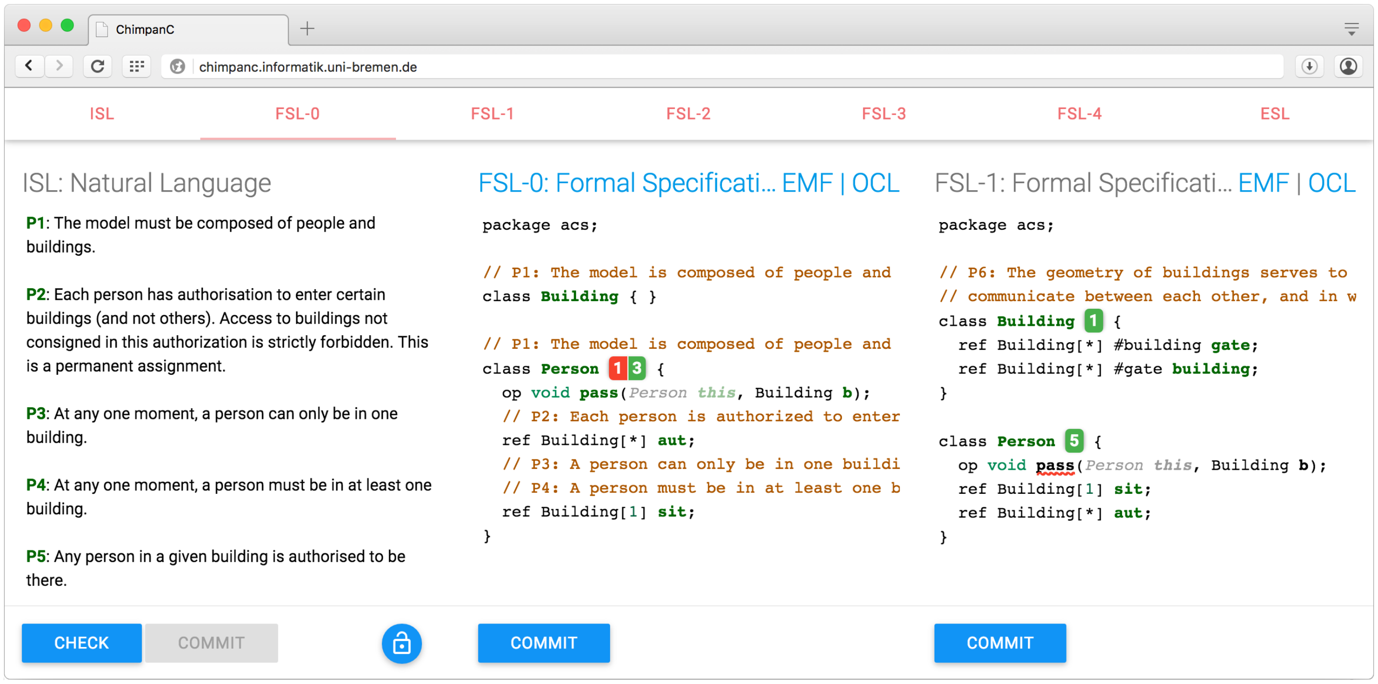
To relate refinement layers users have to provide a project definition file, which declares the refinement structure by a list of layers, each with a type indicator (`isl`, `fsl` or `esl`) for the respective abstraction level and a list of files belonging to this layer.

B. The User Interface

ChImpAnC is realised as a web interface and can either run locally or on a team server. When users open the application in a browser they get presented a multi-column layout representing the different specification layers (Fig. 4). 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.

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. 5).

Inconsistencies are highlighted with red wavy underlines. These include elements (abstract models, attributes, references, operations or parameters) which are unmapped in a refinement (Fig. 6), as well as mismatching mapped types and inconsistent


ChimpanC

chimpanc.informatik.uni-bremen.de

ISL FSL-0 FSL-1 FSL-2 FSL-3 FSL-4 ESL

ISL: Natural Language

P1: The model must be composed of people and buildings.

P2: Each person has authorisation to enter certain buildings (and not others). Access to buildings not consigned in this authorization is strictly forbidden. This is a permanent assignment.

P3: At any one moment, a person can only be in one building.

P4: At any one moment, a person must be in at least one building.

P5: Any person in a given building is authorised to be there.

FSL-0: Formal Specificati… EMF | OCL

```
package acs;

// P1: The model is composed of people and
class Building { }

// P1: The model is composed of people and
class Person 1 3 {
  op void pass(Person this, Building b);
  // P2: Each person is authorized to enter
  ref Building[*] aut;
  // P3: A person can only be in one buildi
  // P4: A person must be in at least one b
  ref Building[1] sit;
}
```

FSL-1: Formal Specificati… EMF | OCL

```
package acs;

// P6: The geometry of buildings serves to
// communicate between each other, and in w
class Building 1 {
  ref Building[*] #building gate;
  ref Building[*] #gate building;
}

class Person 5 {
  op void pass(Person this, Building b);
  ref Building[1] sit;
  ref Building[*] aut;
}
```

Fig. 4. The CHIMPANC user interface.

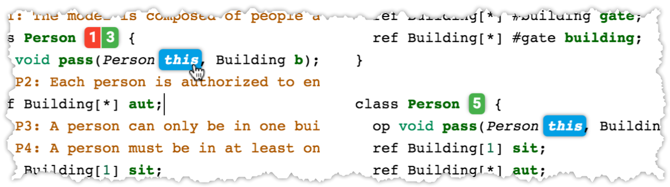
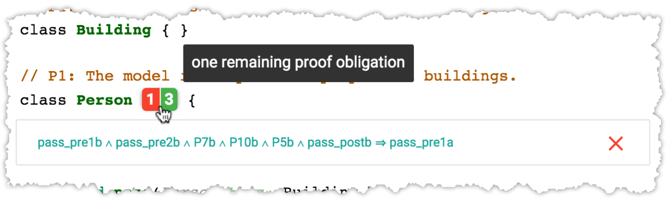

```
1: The model is composed of people a        ref Building[*] #building gate;
s Person 1 3 {                               ref Building[*] #gate building;
 void pass(Person this, Building b);       }
 P2: Each person is authorized to en
f Building[*] aut;                         class Person 5 {
 P3: A person can only be in one bui         op void pass(Person this, Buildin
 P4: A person must be in at least on         ref Building[1] sit;
 Building[1] sit;                            ref Building[*] aut;
```

Fig. 5. Highlighting of mappings

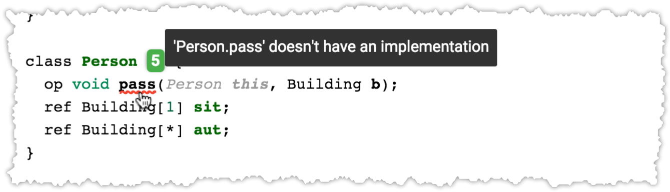

```
class Building { }

// P1: The model                    buildings.
class Person 1 3 {
```

one remaining proof obligation

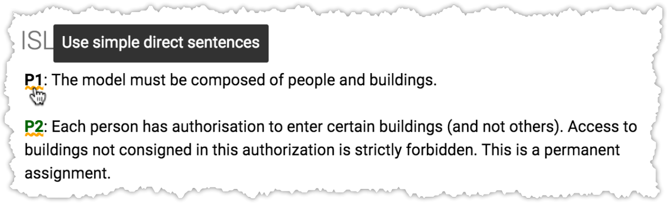
pass_pre1b ∧ pass_pre2b ∧ P7b ∧ P10b ∧ P5b ∧ pass_postb ⇒ pass_pre1a

Fig. 7. Inline display of proof obligations


```
class Person 5
  op void pass(Person this, Building b);
  ref Building[1] sit;
  ref Building[*] aut;
}
```

'Person.pass' doesn't have an implementation

Fig. 6. Highlighting of inconsistencies


ISL Use simple direct sentences

P1: The model must be composed of people and buildings.

P2: Each person has authorisation to enter certain buildings (and not others). Access to buildings not consigned in this authorization is strictly forbidden. This is a permanent assignment.

Fig. 8. A content warning in natural language

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. Conversely, discharged proof obligations appear as a green number (Fig. 7). When the user moves the mouse over a marked element, a tooltip will appear containing information about the inconsistency.

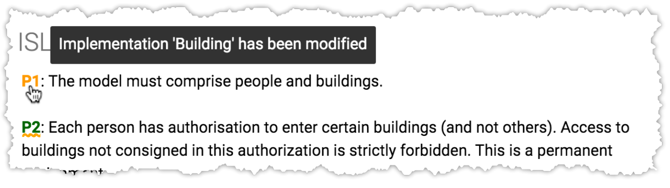
Content warnings are highlighted with orange wavy underlines. These are currently only present in natural language where we automatically rate the quality of refinements, using the techniques from [34]. Again, a detailed description of the warning can be obtained by hovering the mouse over the marked element (Fig. 8).

Finally, change management support is implemented by impacts. 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 be 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. Impact warnings appear as orange elements indicating that user attention is required (Fig. 9).

V. CONCLUSION

This paper 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. Moreover, 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


ISL Implementation 'Building' has been modified

P1: The model must comprise people and buildings.

P2: Each person has authorisation to enter certain buildings (and not others). Access to buildings not consigned in this authorization is strictly forbidden. This is a permanent

Fig. 9. An impact warning

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 Neo4j 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.

In *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 ISL 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.

VI. Acknowledgements

The research reported here was supported by the German Federal Ministry of Education and Research (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, 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," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 2, no. 1, pp. 4–18, 1983.

[4] J.-R. Abrial, "System study: Method and example," http://atelierb.eu/ressources/PORTES/Texte/porte.anglais.ps.gz, 1999.

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

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

[7] M. Jarke, "Requirements tracing," *Communication of the ACM*, vol. 41, no. 12, 1998.

[8] IBM, "Rational DOORS," http://www-03.ibm.com/software/products/en/ratidoor.

[9] G. Fey, D. Große, T. Cassens, C. Genz, T. Warode, and R. Drechsler, "ParSyC: an efficient SystemC parser," in *Workshop on Synthesis And System Integration of Mixed Information technologies*, 2004, pp. 148–154. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.72.7049

[10] F. Karlsruhe, "KaSCPar - Karlsruhe SystemC Parser Suite," 2012, http://www.fzi.de/index.php/de/component/content/article/238-ispe-sim/4350-kascpar-karlsruhe-systemc-parser-suite.

[11] J. Castillo, P. Huerta, and J. I. Martinez, "An open-source tool for SystemC to Verilog automatic translation," *Latin American Applied Research*, vol. 37, no. 1, pp. 53–58, 2007. [Online]. Available: http://www.scielo.org.ar/scielo.php?script=sci_arttext&amp;pid=S0327-07932007000100011

[12] C. Brandolese, P. Di Felice, L. Pomante, and D. Scarpazza, "Parsing SystemC: an open-source, easy-to-extend parser," in *IADIS International Conference on Applied Computing*, 2006, pp. 706–709.

[13] D. Berner, J.-P. Talpin, H. Patel, D. A. Mathaikutty, and S. Shukla, "SystemCXML: An extensible SystemC front end using XML," in *Proceedings of the Forum on Specification and Design Languages*, 2005, pp. 405–409. [Online]. Available: http://i-tecs.fr/ecsi/libraryV1/uploads/6-CSD21_paper.pdf

[14] C. Genz and R. Drechsler, "Overcoming limitations of the SystemC data introspection," in *Proceedings of the Conference on Design, Automation and Test in Europe*, 2009, pp. 590–593. [Online]. Available: http://dl.acm.org/citation.cfm?id=1874764

[15] M. Moy, F. Maraninchi, and L. Maillet-Contoz, "Pinapa: An extraction tool for systemc descriptions of systems-on-a-chip," in *Conference on Embedded software*, 2005, pp. 317–324. [Online]. Available: http://dl.acm.org/citation.cfm?id=1086286

[16] D. Große, R. Drechsler, L. Linhard, and G. Angst, "Efficient automatic visualization of systemc designs," in *Forum on Specification & Design Languages*, 2003, pp. 646–658.

[17] J. Stoppe, R. Wille, and R. Drechsler, "Data extraction from SystemC designs using debug symbols and the SystemC API," in *IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*. IEEE, 2013, pp. 26–31.

[18] B. Meyer, "Applying 'design by contract'," *Computer*, vol. 25, no. 10, pp. 40–51, Oct 1992.

[19] L. Benvenuti, A. Ferrari, E. Mazzi, and A. L. S. Vincentelli, *Contract-Based Design for Computation and Verification of a Closed-Loop Hybrid System*. Springer, 2008, pp. 58–71. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-78929-1_5

[20] P. Chalin, J. R. Kiniry, G. T. Leavens, and E. Poll, *Beyond Assertions: Advanced Specification and Verification with JML and ESC/Java2*. Springer, 2006, pp. 342–363. [Online]. Available: http://dx.doi.org/10.1007/11804192_16

[21] P. Baudin, P. Cuoq, J.-C. Filliâtre, C. Marché, B. Monate, Y. Moy, and V. Prevosto, *ACSL: ANSI/ISO C Specification Language Version 1.11*, CEA LIST and INRIA. [Online]. Available: http://frama-c.com/download/acsl.pdf

[22] F. L. Bauer, M. Broy, R. Gnatz, W. Hesse, B. Krieg-Brückner, H. Partsch, P. Pepper, and H. Wössner, "Towards a wide spectrum language to support program specification and program development," *ACM SIGPLAN Notices*, vol. 13, no. 12, pp. 15–24, 1978.

[23] J.-R. Abrial, J.-R. Abrial, and A. Hoare, *The B-book: assigning programs to meanings*. Cambridge University Press, 2005.

[24] J.-R. Abrial, M. Butler, S. Hallerstede, T. S. Hoang, F. Mehta, and L. Voisin, "Rodin: an open toolset for modelling and reasoning in Event-B," *International journal on software tools for technology transfer*, vol. 12, no. 6, pp. 447–466, 2010.

[25] B. Dion and J. Gartner, "Efficient development of embedded automotive software with IEC 61508 objectives using SCADE drive," in *VDI 12th International Conference: Electronic Systems for Vehicles*. VDI, 2005, pp. 1427–1436.

[26] 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, 2002, no. 2263, pp. 42–68.

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

[28] D. Steinberg, F. Budinsky, E. Merks, and M. Paternostro, *EMF: Eclipse Modeling Framework*. Pearson Education, 2008.

[29] A. Lanusse, Y. Tanguy, H. Espinoza, C. Mraidha, S. Gerard, P. Tessier, R. Schnekenburger, H. Dubois, and F. Terrier, "Papyrus UML: an open source toolset for MDA," in *Proc. of the Fifth European Conference on Model-Driven Architecture Foundations and Applications (ECMDA-FA 2009)*. Citeseer, 2009, pp. 1–4.

[30] 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.

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

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

[33] 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, 2000, pp. 9–14.

[34] 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*, 2014.