Formal Specification Level: Towards Verification-driven Design Based on Natural Language Processing

Rolf Drechsler*† Mathias Soeken*† Robert Wille*

*Institute of Computer Science, University of Bremen, 28359 Bremen, Germany
†Cyber-Physical Systems, DFKI GmbH, 28359 Bremen, Germany
{drechsle,msoeken,rwille}@informatik.uni-bremen.de

***Abstract*—The steadily increasing complexity of the design of embedded systems led to the development of both an elaborated design flow that includes various abstraction levels and corresponding methods for synthesis and verification. However, until today the initial system specification is provided in natural language which is manually translated into a formal implementation e.g. at the *Electronic System Level* (ESL) by means of SystemC in a time-consuming and error-prone process.**

In this paper, we envision a design flow which incorporates a *Formal Specification Level* (FSL) thereby bridging the gap between the informal textbook specification and the formal ESL implementation. Modeling languages such as UML or SysML are envisaged for this purpose. Recent accomplishments towards this envisioned design flow, namely the automatic derivation of formal models from natural language descriptions, verification of formal models in the absence of an implementation, and code generation techniques, are briefly reviewed.

I. Introduction

Being composed of up to several billion components, the design of embedded systems is one of the most complex problems people are facing today. While it was possible to fully design such systems gate by gate on the drawing table 40 years ago, this procedure has become intractable due to the ever increasing complexity. As a consequence, elaborated design flows have been developed over the last decades in which several levels of abstraction are considered.

Today, a design flow as briefly illustrated in Fig. 1(a) is applied. The initial starting point is given by means of a specification which is usually provided in terms of a text book description, however, in order to perform even the simplest automatic synthesis techniques, a formal representation of the specification is required. For this purpose, an initial implementation is generated at the *Electronic System Level* (ESL) using high-level programming languages such as SystemC. This system level description enables the execution and simulation of the desired design, but still hides details concerning a precise realization in both hardware and software. From this description, the system model is consecutively refined in successive steps leading to descriptions at the *Register Transfer Level* (RTL), the *gate level*, and the *physical level*. At the end of this process, the resulting chip is sent to a chip manufacturer.

As embedded systems are often employed in safety critical systems such as avionic, automotive, and medical applications, ensuring the correctness is of high importance. For this purpose, usually each transformation from one abstraction level to the next refinement is checked for equivalence. But due to the absence of a formal description at the specification level, automatic verification techniques are not applicable for the comparison with the system level. Further, as the system level representation is manually derived from the textual specification, this step is particularly prone to errors and mistakes.

So far property checking is applied to address this issue by extracting properties from the specification in terms of temporal logic expressions which can subsequently be checked by using algorithms known model checkers [1]. Further techniques called coverage detection exist that can automatically determine whether enough properties have been written, i.e. whether the full behavior is considered by all properties [2]. However, the main obstacle remains, i.e. the specification is provided in natural language and a formal representation needs to be manually derived from it for further processing. Motivated by this, researchers started working on closing the gap between the informal textbook specifications and the respective ESL implementation [3], [4].

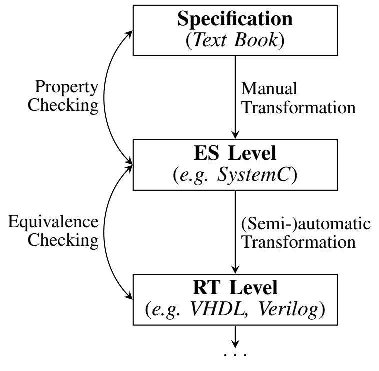
In this work, we envision a new design flow which exploits recent achievements in this area. For this purpose, we propose two major extensions.

First, we follow the steady strive for higher levels of abstraction and enrich the specification itself by formal description means. Modeling languages such as the *Unified Modeling Language* (UML) [5] or the *System Modeling Language* (SysML) [6] combined with constraints provided in the *Object Constraint Language* (OCL) [7] provide proper syntax and semantic for this purpose.[1] While these description means remain abstract enough for the specification level, their formal description enables (semi-)automatic verification and code construction. As a result, crucial design flaws can already be detected at the specification level and thus in the absence of a precise implementation.

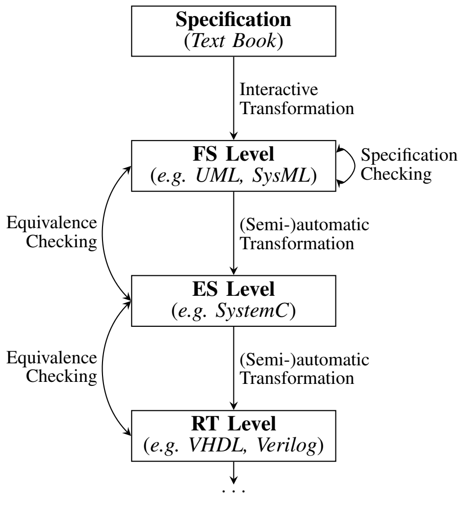
Second, initial solutions are applied to automatically derive the respective UML/OCL descriptions from the natural language specification. Recent achievements in the area of natural language processing [8], information extraction [9], and knowledge representation [10] are exploited for this purpose. In fact, already simple grammatical analyses enable e.g. the derivation of (1) basic components of a system (which can be derived from nouns in a sentence), (2) their functions (which can be derived from verbs in a sentence), and (3) attributes (which can be derived from adjectives in a sentence).

Having such methods, we envision a design flow which includes a *Formal Specification Level* (FSL) as shown in Fig. 1(b). This flow enables to (semi-)automatically derive formal descriptions from a given specification provided in natural language. Formal methods are applied to this description to verify the correctness of the design prior to an implementation. If all checks passed, code skeletons for synthesis and formal properties for verification are extracted for further usage within the remaining stages of the established design flow.

[1] In the following, we focus on UML/OCL, while the general concepts are similarly applicable to other modeling languages as well.



(a) Conventional design flow



(b) Envisioned design flow

Fig. 1. Conventional and envisioned design flow

In the remainder of this paper, the general ideas and first accomplishments towards this envisioned design flow are presented. The following section briefly introduces the necessary background to keep the paper self-contained. Afterwards, Section III outlines the proposed extension to the overall design flow in detail. The respective steps for mapping a natural language specification to a formal model, checking the correctness of that formal model, and transforming the formal model into an implementation are then outlined in Section IV, Section V, and Section VI, respectively. Finally, remaining challenges to be addressed are discussed and the paper is concluded in Section VII.

II. Background

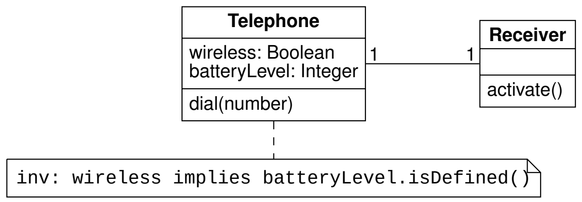
In this work, the *Unified Modeling Language* (UML) is applied to represent the code skeletons and test cases which are semi-automatically derived from natural language. Besides that, we also exploit language processing tools. To keep the paper self-contained, the underlying concepts of UML and the applied tools are briefly reviewed in the following.

A. Unified Modeling Language

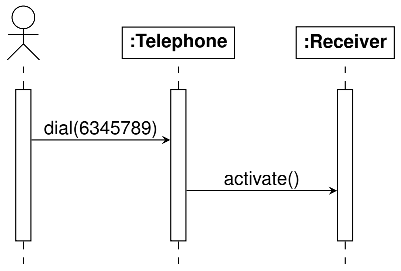
In this section, we briefly review the basic UML concepts which are considered in this work. A detailed overview of the UML is provided in [5].

1) Class Diagrams: A UML *class diagram* is used to represent the structure of a system. The main component of a class diagram is a *class* that describes an atomic entity of the model. A class itself consists of *attributes* and *operations*. Attributes describe the information which is stored in the class (e.g. member variables). Operations define possible actions that can be executed e.g. in order to modify the values of attributes. Classes can be set into relation via *associations*. The type of a relation is expressed by *multiplicities* that are added to each association-end. Class diagrams can be extended by constraints in the *Object Constraint Language* (OCL) such as invariants that further restrict the attribute values.

Example 1: Fig. 2(a) shows a UML class diagram specifying a simple telephone. The class diagram consists of the two classes *Telephone* and *Receiver*. The class *Telephone* has an attribute *wireless* of type *Boolean*. The receiver is related to the telephone which is expressed by an association. As expressed by the multiplicities, each telephone has one receiver and vice versa. Both classes have an operation, i.e. the telephone can dial a number and the receiver can be activated. The single OCL invariant in the diagram states that if a telephone is wireless, its battery level needs to be defined.



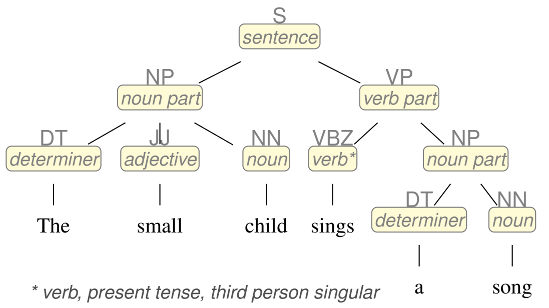
(a) UML class diagram



(b) UML sequence diagram

Fig. 2. UML class and sequence diagram

2) Sequence Diagrams: The dynamic flow caused by operation calls can be visualized by sequence diagrams. Sequence diagrams offer the possibility to represent particular scenarios (i.e. behavior) based on the model provided by the class diagram. Hence, several sequence diagrams exist for a given class diagram. In the sequence diagram, instances of the classes, i.e. objects, are extended by *life lines* that express the time of creation and destruction in the scenario. Arrows indicate operations that are called on an object, and are drawn from the caller to the callee. Besides objects also actors from the outside environment can be part of the sequence diagram.



(a) Phrase structure tree

det(child-*3*, The-*1*)
amod(child-*3*, small-*2*)
nsubj(sings-*4*, child-*3*)
root(ROOT-*0*, sings-*4*)
det(song-*6*, a-*5*)
dobj(sings-*4*, song-*6*)

(b) Typed dependencies

Fig. 3. Application of the Stanford Parser

Example 2: A sequence diagram is depicted in Fig. 2(b). In that scenario, first a number is dialed from an actor in the outside environment, before the telephone activates the receiver.
In this work, class diagrams and sequence diagrams are applied to represent the semi-automatically determined code skeletons and test cases, respectively.

B. Stanford Parser

The Stanford Parser is an open source software compilation published by the Stanford *Natural Language Processing* (NLP) Group [8]. It parses sentences in different languages and returns a *Phrase Structure Tree* (PST) representing the semantic structure of the sentence. A PST is an acyclic tree with one root vertex representing a given sentence. Non-terminal and terminal vertices (i.e. leafs) represent the grammatical structure and the atomic words of this sentence, respectively. A simple PST for the sentence “The small child sings a song” is given by means of Fig. 3(a). As can be seen all leafs are connected to distinct vertices that classify the *tag* of the respective word, e.g. nouns and verbs. These word tags are further grouped and connected by other vertices labeled with a tag classifying a part of the sentence, e.g. as noun parts or verb parts. The classifier tags are abbreviated in the PST, however, in Fig. 3(a) the full classifier is annotated to the vertices. For details on how a PST is extracted from a sentence, the reader is referred to [11].

Besides the PST, the Stanford Parser also provides *typed dependencies* [12] which are very helpful in NLP. Typed dependencies are tuples which describe the semantic correlation between words in the sentence. Fig. 3(b) lists all typed dependencies for the sentence considered in Fig. 3(a). For example, the nouns are assigned their articles using the **det** relation. Note that the numbers after the word refer to the position in the text, which is necessary if a word occurs more than once in a sentence. Two further important relations are **nsubj** and **dobj** that allow for the extraction of the typical *subject-verb-object* form. In this case it connects the verb *sings* with both its subject and object.

In this work, the Stanford Parser is applied to process the structure of the sentences describing a scenario.

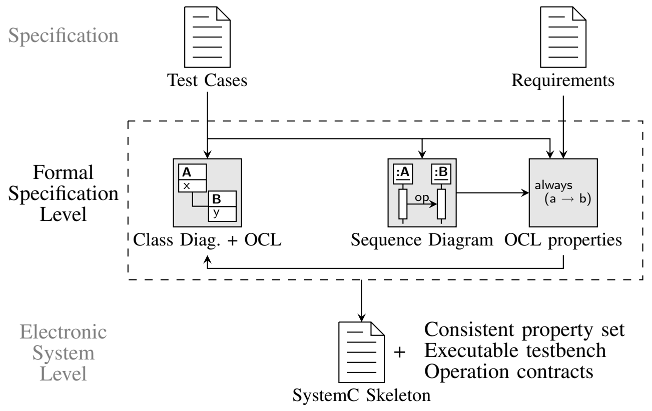


Fig. 4. Overview of the Formal Specification Level

C. WordNet

WordNet [10], developed at Princeton University, is a large lexical database of English that is designed for use under program control. It groups nouns, verbs, adjectives, and adverbs into sets of cognitive synonyms, each representing a lexicalized concept. Each word in the database can have several *senses* that describe different meanings of the word. In total, WordNet consists of more than 90,000 different word senses, and more than 166,000 pairs that connect different senses with a semantic meaning.

Further, each sense is assigned a small description text which makes the precise meaning of the word in that context obvious. Frequency counts provide an indication of how often the word is used in common practice. The database does not only distinguish between the word forms noun, verb, adjective, and adverb, but further categorizes each word into sub-domains. Those categories are e.g. *artifact*, *person*, or *quantity* for a noun.

In this work, WordNet is applied to determine the semantics of the sentences describing a scenario.

III. The Formal Specification Level

Fig. 4 provides a more detailed view on the proposed extension for the envisioned design flow. The main goal is to (semi-)automatically derive an ESL-implementation in SystemC[2] from a (textbook) specification provided in natural language. Given natural language test cases and requirements from the specification, an initial SystemC implementation, an executable testbench for simulation, and operation contracts (pre- and post-conditions as motivated by *Design-by-Contract* [13]) are (semi-)automatically generated. For this purpose, the Formal Specification Level as shown in Fig. 1(b) and detailed in Fig. 4 is introduced as a new abstraction level which includes three stages.

In the first stage (cf. Section IV), the test cases and the requirements are mapped from their natural language description into a formal representation by means of UML/OCL. NLP techniques are exploited in order to extract the desired information. More precisely, the following steps are conducted in this first stage:

- *Determine the structure of the design*
  Using e.g. a grammatical analysis, the basic components of the considered system are derived from the natural language specification. From the resulting information,

[2]Note that SystemC is just a representative for any high-level object-oriented hardware description language and can readily be replaced.

a UML class diagram is created which provides a first formal description of the structure for the considered design.

- *Determine the behavior of the design*
  In a similar fashion, execution sequences are derived from the natural language specification. They are used to create UML sequence diagrams representing certain scenarios and thus behavior to be considered in the design.
- *Determine the properties of the design*
  After having both the structure and the scenarios, the requirements of the specification can be considered in detail. From them, formal properties which need to be satisfied by the design are derived and represented in terms of OCL expressions.

As a result, the first stage leads to a formal description of the desired system in terms of UML/OCL.

In the second stage (cf. Section V), this formal description is used to conduct initial checks for correctness. This e.g. includes consistency checks such as checking whether it is possible to instantiate the desired system considering all constraints and requirements, but also first behavioral checks such as checking whether it is possible to reach a prohibited state. This allows for the detection of design flaws already in very early design steps, even in the absence of a precise implementation.

In the third stage (cf. Section VI), after all checks have passed and no errors have been determined, a skeleton for the system level implementation as well as corresponding testbenches are derived.

In the next sections, first accomplishments with respect to these stages of the FSL are illustrated.

IV. Mapping Natural Language Specifications to the Formal Specification Level

The first stage addresses the (semi-)automatic determination of a formal representation describing the structure, the behavior, and the properties of a system that is specified in natural language. First accomplishments for the former two aspects have been presented in [14] and are reviewed in the following two sub-sections. Afterwards, initial ideas on the property determination are presented.

A. Determine the Structure of the Design

Test cases inside a specification are written in a very specific way, i.e. by using short sentences which describe the elementary steps of a scenario. From this, much information can already been determined automatically. As an example, consider the following test case describing how a user is placing a telephone call:

A caller picks up the receiver from a telephone.
The caller dials the number 6-345-789.
The telephone places a call.

Fig. 5 illustrates that already from these three sentences a significant amount of structural information can be extracted: Since telephone and receiver are object nouns, it can be concluded that they represent components of the considered system (to be represented by classes). Preceded adjectives (such as wireless) substantiate objects and, thus, shall be added as attributes to the corresponding class. Verbs correlate to operations which can be invoked by components or actors.

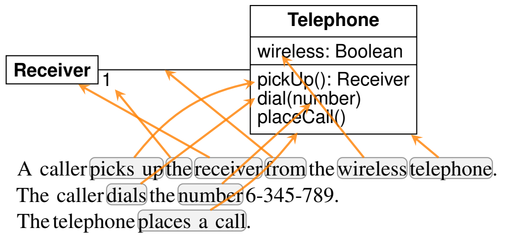

Telephone
wireless: Boolean
pickUp(): Receiver
dial(number)
placeCall()
Receiver
1
A caller picks up the receiver from the wireless telephone.
The caller dials the number 6-345-789.
The telephone places a call.

Fig. 5. Determine the structure of the design

Moreover, prepositions help to determine relations between classes. For example, the receiver from a telephone does not only imply a relation due to the preposition from but also indicates that a telephone can only have one receiver due to the definite article the.

Recent progress in the development of NLP technologies enables to extract much of these information in a (semi-)automatic manner. More precisely, NLP parsers (e.g. the one presented in [11]) are able to decompose a sentence in terms of a *phrase structure tree* (PST) which assigns each atomic word to a syntactic word type (such as noun, verb, or adjective) and also groups words into larger sub-parts of the sentence (cf. Section II-B).

However, sometimes the syntactical and grammatical information alone is not sufficient. For example in the first sentence from Fig. 5, three nouns are identified in the PST, i.e. caller, receiver, and telephone but only for the two latter ones classes need to be created. This information cannot be derived from the PST. Hence, we are additionally making use of an ontology that allows for a further semantical classification of the words. A look into a word database such as WordNet [10] reveals that the first noun is of class *person* whereas the other nouns are listed as *artifacts*. Hence, the caller is treated as an actor of the system.

Overall, exploiting these NLP technologies, a UML class diagram formally representing the structure of the considered system can automatically be determined in many cases. However, since the textual description always can contain ambiguities, manual interactions with the design engineer cannot entirely be excluded leading to a (semi-)automatic and assisted approach as evaluated in [14].

B. Determine the Behavior of the Design

Besides the structure, test cases of a specification also provide information on certain scenarios, i.e. a sequence of actions to be conducted by the considered design. Formally, such scenarios can be represented by sequence diagrams as introduced in Section II-A2 which represent certain behavior of the design. The mapping from the natural description of a test case to a sequence diagram can be performed in a similar way as done above for determining the structure.

More precisely, each verb in a sentence can be mapped to a corresponding operation call. The caller and a possible callee can be determined by the subject and the object, respectively. Structural information of the system derived beforehand are exploited for this purpose. Fig. 6 illustrates how a sequence diagram is built from the exemplary test case.

However, also here some obstacles need to be addressed. For example, instead of parameters for a function (such as number in Fig. 5), now the actual value for an operation call needs to be fetched (such as 6-345-789 in Fig. 6). Furthermore, objects and actors that reoccur in successive sentences need to be

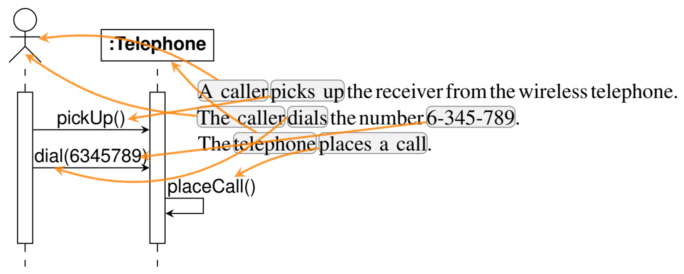

:Telephone
pickUp()
dial(6345789)
placeCall()
A caller picks up the receiver from the wireless telephone.
The caller dials the number 6-345-789.
The telephone places a call.

Fig. 6. Determine the behavior of the design

determined as such and linked to their original occurrence. As an example, The caller in the second sentence refers to A caller from the first sentence, i.e. they refer to the same actor in the sequence diagram.

For this purpose, again the PST is analyzed. As an example, due to the definite article The in the third sentence, it becomes clear that the same telephone as in the first sentence is addressed. If instead an indefinite article had been used as the determiner for telephone, a new object would have been instantiated for the sequence diagram.

C. Determine the Properties of the Design

The scenarios derived in the last step describe sequences of actions with precise simulation parameters. This helps testing the basic functionality of the desired system, but is insufficient to actually *proof* its correctness. For example, while the basic functionality of a traffic light controller can easily be validated by some test case scenarios, general (safety-critical) requirements such as "the pedestrian light and the car light are never supposed to be both green at the same time" require a more exhaustive approach. For such purposes, properties are usually defined which afterwards are checked by model checkers.[3]

So far, such properties are mainly manually derived from the textual specification. However, also here a systematic approach can be applied [15]. For example, taking the "never green at the same time"-requirement from above, the elementary sub-terms of the expression can be detected and linked to their respective model elements leading to:

Sub-sentence	Model element
'pedestrian light'	`controller.pedLight`
'car light'	`controller.carLight`
'show green'	x `== true`

With the aid of the adverbs in the requirement, these code parts can be joined together to form an invariant such as

```
inv: not (controller.pedLight and controller.carLight).
```

Besides that, it is also possible to generalize properties from test cases when they obey a certain structure. For example, in the context of *Behavior Driven Development* (BDD) the structure of a test case is often given by a *Given* A, *When* B, *Then* C template [16]. Since A corresponds to environment constraints, B corresponds to the antecedent, and C corresponds to the consequent of a property, formal properties can be generalized from such test cases [17].

[3]Note that, in the following we are using the terms properties and invariants almost synonymously as "properties" are more common in the context of formal verification whereas "invariants" is a common term used in the context of modeling.

V. Checking Correctness at the Formal Specification Level

After the first stage, a formal representation has been derived which is sufficient to provide information about the structure, the behavior, and the properties of the desired system while still hiding precise implementation details. In the second stage, this representation enables to conduct correctness checks of the design in the absence of an implementation. For this purpose, approaches presented in [18], [19], [20], [21] for static verification, presented in [22] for invariant elimination, and presented in [23], [24] for dynamic verification can be applied. In [25], also first debugging approaches have been introduced.

A. Verification of Static Aspects

Having a formal representation of the design does not necessarily imply that a working implementation can be generated from it. In fact, the formal model may inherit constraints which contradict each other. As a result, no valid instantiation would be possible and any implementation would be erroneous from scratch. The FSL enables to detect such errors before any code is written.

Approaches introduced e.g. in [18], [19], [20], [21] can be utilized for this purpose. They take the obtained UML diagram (representing the structure) together with the properties (which are encoded as OCL invariants) and automatically perform the above described *consistency checks*. Besides enumerative methods [20], also elaborated formal approaches have been proposed in the recent past [21]. Considering the abstract description of the models (usually, no complex data-structures are applied), particularly the latter approaches are applicable to quite significantly complex designs.

B. Invariant Removal

At the FSL, invariants are a proper description mean to represent properties the design has to satisfy. However, when it comes to verification they may cause unnecessary overhead. Since invariants are assumed globally, i.e. for each possible system state of the system, they have to be considered all the time. Even if only a certain functionality of a design is under verification, *invariants* of the entire model have to be assumed additionally.

An alternative to prevent this overhead has been proposed in [22]. Here, invariants are iteratively removed and replaced instead with a smaller set of pre- and post-conditions for certain operations. This enables to entirely eliminate all invariants without changing the semantics of the model. Since additionally, pre- and post-conditions only have to be considered locally when the corresponding function is called, this reduces the overhead.

Furthermore, invariant elimination enables a design flow in which the implementation of different operations can be conducted by different developers. Then, the respective sub-teams do not have to globally consider all the invariants anymore, but just the local pre- and post-conditions of the corresponding operation.

C. Verification of Dynamic Aspects

Finally, also the dynamic behavior can be verified at the FSL. This is possible due to the above-mentioned pre- and post-conditions of operations which enable a descriptive representation of the behavior, without giving a precise implementation. A pre-condition describes in which states an operation can be called, while the post-condition describes the effect an operation has on that system state. These conditions may be specified directly from the designer or are determined by the invariant elimination step described above.

Any model where its operations are enriched with pre- and post-conditions can be transformed into an instance similar to *Bounded Model Checking* (BMC) [26] and, therefore, allows for addressing certain dynamic verification tasks. In fact, similar to verification at the implementation level, operation sequences can be determined that lead e.g. to bad states, good states, live locks, or dead locks [23]. Utilizing these techniques, again, errors can be detected before any code is written.

VI. Mapping from Formal Specification Level to the Electronic System Level

Finally, the formally modeled and verified design shall be implemented in a proper ESL-language so that it can be further refined using the established design flow. Also in this final stage of the FSL the formal representation can be exploited.

In fact, the corresponding UML/OCL descriptions allow for a generation of code parts for the implementation process. This includes

- code stubs generated from class diagrams (Section IV-A),
- an executable testbench generated from the sequence diagrams (Section IV-B),
- generalized properties from the parameterized test cases (Section IV-C),
- a consistent property set (Section V-A),
- and contracts for the operations of a class (Section V-B).

Further, the verification of dynamic aspects plays a significant role in the transition from the FSL to the ESL. As briefly discussed in Section V-C, all dynamic aspects, i.e. the interaction of the components, can be checked in the absence of a precise implementation. As an example, it can be ensured that the model is deadlock-free or that all operations can be reached from given initial states. Hence, after the implementation phase, it is sufficient to check whether the implementation of each single operation adheres correctly to its contracts. That is, assuming the pre-condition and executing the code must imply the post-conditions. Since the verification of the operations can be performed locally without considering the whole system, verification effort can be decreased.

VII. Conclusion

In this paper, we envisioned a new design flow which includes an FSL representing the desired design using modeling languages such as UML or SysML combined with constraints provided in languages such as OCL. The proposed flow bridges the gap between the natural language textbook specification and the formal ESL implementation. We illustrated that first accomplishments towards the envisioned design flow have already been made: NLP techniques are available to derive formal descriptions of natural language specifications, verification approaches based on modeling languages allow to detect design errors prior to a precise implementation, and code generation techniques can be applied to generate code stubs, executable testbenches, etc.

Acknowledgments

This work was supported by the German Research Foundation (DFG) within the Reinhart Koselleck project DR 287/23-1.

References

[1] E. M. Clarke, Jr., O. Grumberg, and D. A. Peled, *Model Checking*. Cambridge, MA, USA: MIT Press, 1999.

[2] D. Große and R. Drechsler, *Quality-Driven SystemC Design*. Dordrecht, Heidelberg, London, New York: Springer, Dec. 2009.

[3] R. Drechsler, "Quality-driven Design of Embedded Systems based on Specification in Natural Language," in *EUROMICRO Symp. on Digital System Design*, 2011.

[4] I. G. Harris, "Extracting design information from natural language specifications," in *Design Automation Conference*, 2012, pp. 1256–1257.

[5] J. Rumbaugh, I. Jacobson, and G. Booch, *The Unified Modeling Language reference manual*. Essex, UK: Addison-Wesley Longman, Jan. 1999.

[6] T. Weilkiens, *Systems Engineering with SysML/UML: Modeling, Analysis, Design*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., Feb. 2008.

[7] J. Warmer and A. Kleppe, *The Object Constraint Language: Precise modeling with UML*. Boston, MA, USA: Addison-Wesley Longman, Mar. 1999.

[8] D. Jurafsky and J. H. Martin, *Speech and Language Processing*. Pearson Prentice Hall, 2008.

[9] J. R. Cowie and W. G. Lehnert, "Information Extraction," *Communications of the ACM*, vol. 39, no. 1, pp. 80–91, Jan. 1996.

[10] G. A. Miller, "WordNet: A Lexical Database for English," *Communications of the ACM*, vol. 38, no. 11, pp. 39–41, Nov. 1995.

[11] D. Klein and C. D. Manning, "Accurate Unlexicalized Parsing," in *Annual Meeting of the Association for Computational Linguistics*, July 2003, pp. 423–430.

[12] M.-C. de Marneffe, B. MacCartney, and C. D. Manning, "Generating Typed Dependency Parses from Phrase Structure Parses," in *Int'l Conf. on Language Ressources and Evaluation*, May 2006, pp. 449–454.

[13] B. Meyer, J.-M. Nerson, and M. Matsuo, "EIFFEL: Object-Oriented Design for Software Engineering," in *European Software Engineering Conference*, ser. Lecture Notes in Computer Science, H. K. Nichols and D. Simpson, Eds., vol. 289. Springer, Sept. 1987, pp. 221–229.

[14] M. Soeken, R. Wille, and R. Drechsler, "Assisted Behavior Driven Development Using Natural Language Processing," in *Int'l. Conf. on Objects, Models, Components, Patterns*, May 2012.

[15] H. M. Le, D. Große, and R. Drechsler, "From Requirements and Scenarios to ESL Design in SystemC," in *Int'l Symp. on Electronic System Design*, Dec. 2012.

[16] D. North, "Behavior Modification: The evolution of behavior-driven development," *Better Software*, vol. 8, no. 3, Mar. 2006.

[17] M. Diepenbeck, M. Soeken, D. Große, and R. Drechsler, "Behavior Driven Development for Circuit Design and Verification," in *IEEE Intl'l High Level Design Validation and Test Workshop*, Nov. 2012.

[18] D. Jackson, *Software Abstractions: Logic, Language, and Analysis*. Cambridge, MA, USA: MIT Press, Apr. 2006.

[19] J. Cabot, R. Clarisó, and D. Riera, "Verification of UML/OCL Class Diagrams using Constraint Programming," in *IEEE Int'l. Conf. on Software Testing Verification and Validation Workshop*, Apr. 2008, pp. 73–80.

[20] M. Gogolla, M. Kuhlmann, and L. Hamann, "Consistency, Independence and Consequences in UML and OCL Models," in *Tests and Proofs*. Springer, July 2009, pp. 90–104.

[21] M. Soeken, R. Wille, M. Kuhlmann, M. Gogolla, and R. Drechsler, "Verifying UML/OCL models using Boolean satisfiability," in *Design, Automation and Test in Europe*, Mar. 2010, pp. 1341–1344.

[22] M. Soeken, R. Wille, and R. Drechsler, "Eliminating Invariants in UML/OCL Models," in *Design, Automation and Test in Europe*, Mar. 2012, pp. 1142–1145.

[23] ——, "Verifying Dynamic Aspects of UML Models," in *Design, Automation and Test in Europe*, Mar. 2011, pp. 1077–1082.

[24] J. Cabot, R. Clarisó, and D. Riera, "Verifying UML/OCL Operation Contracts," in *Integrated Formal Methods*, ser. Lecture Notes in Computer Science, M. Leuschel and H. Wehrheim, Eds., vol. 5423. Springer, Feb. 2009, pp. 40–55.

[25] R. Wille, M. Soeken, and R. Drechsler, "Debugging of Inconsistent UML/OCL Models," in *Design, Automation and Test in Europe*, Mar. 2012, pp. 1078–1083.

[26] A. Biere, A. Cimatti, E. M. Clarke, O. Strichman, and Y. Zhu, "Bounded model checking," *Advances in Computers*, vol. 58, pp. 117–148, 2003.