SPECifIC — A New Design Flow for Cyber-Physical Systems

Christoph Lüth, Serge Autexier, Dieter Hutter, Mathias Soeken, Robert Wille, Rolf Drechsler
DFKI Bremen
Cyber-Physical Systems Research Group
Bremen, Germany

***Abstract*—In this position paper, we propose a new design flow for cyber-physical systems which builds on our previous experiences in both the hardware and software domain. Its defining features are the integration of natural language processing, the formal specification level which allows an abstract description of the system's behaviour, and a comprehensive functional change management throughout. We introduce the design flow and its three levels of abstraction by example, and argue why it is particularly suited for the development of cyber-physical systems.**

I. Introduction

Due to their very characteristics, the development of correct cyber-physical systems is a greater challenge than those of traditional embedded or software systems. The reasons for this complexity include:

- during their operational time, the working *environment* of CPS *changes*, often in an unforeseen fashion;
- because of this, the (formal) *specification* of the CPS remains *unclear* initially;
- and hence, our development process needs to be able to *handle frequent changes* gracefully;
- CPS often operate in close collaboration with humans, leading to obvious *legal issues* with regards to health and safety concerns;
- besides *distinguishing* which parts of a CPS are realised in *software and hardware*.

In this position paper, we aim to address these issues and sketch how the Cyber-Physical Systems research group at DFKI Bremen aims to tackle them in the nearer future. To this end, we propose a new, comprehensive design flow informed by our previous experiences.

II. Our Background

The background of our group includes formal development and modelling in both the hardware and software domain, combined with a long-standing experience in change management.

- *Hardware verification* has been a focus of the work in the computer architecture group at the University of Bremen for a number of years. In a large number of projects, often joint with industrial partners such as NXP, Infineon, or Intel Mobile Solutions, formal tools have been developed for the design flow. These include approaches based on BDDs [1], [2], SAT solvers, test pattern generation, or their combinations [3], [4], which are targeted at the lower and more concrete levels of the development process, or tools for model-checking SystemC specifications [5].
- The concept of a development graph [6], pioneered by our group in the 90s, is a key concept for *change management*. The generic DocTIP system [7] and its more specific instantiation SmartTies [8] supported the semantic analysis of structured documents by the declarative specification of the semantic structure (document models), and rules describing the impact of changes in the structure.
- Work on *software verification*, in particular in the robotics domain [9], [10], used the interactive theorem prover Isabelle as well as automatic tools, focusing on the C programming language [11] and also integrating change management into the verification process [12]. The robotics domain is a particularly good representative for cyber-physical systems as it combines a rich physical environment requiring expressive modelling with sophisticated control algorithms, unlike traditional embedded systems.

III. Addressing the Challenges

We aim to address the challenges laid out in Sect. I by the following three measures: the integration of *natural language* specifications; the introduction of a new abstract system description level called the *formal specification level* (FSL); and *functional change management* throughout, which allows us both to react to changes in specifications as well as to reuse existing developments.

A. Natural Language Specifications

We explicitly consider natural language specifications, from which we synthesise formal specifications (early work included *e.g.* extracting SysML requirement diagrams with formal OCL specifications from English specifications). One advantage of natural language is that it is more inclusive than any formal specification language, and subsequently can be understood by far more stakeholders of the development process. Moreover, this approach bridges the notoriously difficult first step from informal to formal (safety) requirements; if we can extract the latter automatically from the former, the gap narrows and we can concentrate attention, in particular in the reviewing process, on the more easily understood informal specification. This addresses the issues of unclear

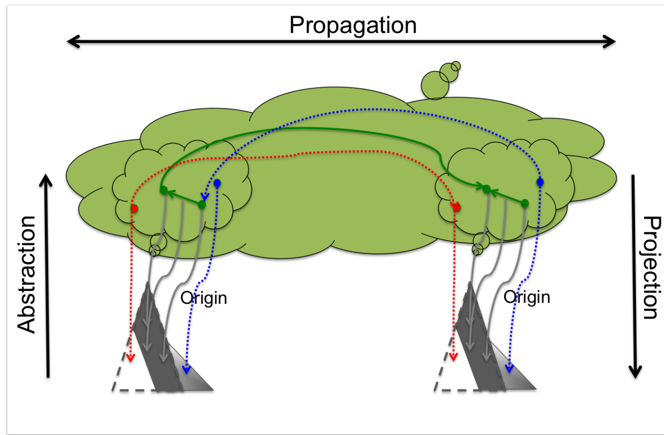

Fig. 1. Functional change management: from the documents (lower row), we *abstract* a semantics (symbolised by the cloud), which allows us to *propagate* the impact of changes and *project* them back to the documents, by keeping track of their origin.

specifications above, but also — by making specifications more readily understood and easy to review — legal issues.

B. The Formal Specification Level

The *formal specification level* models the functional aspects of the system on an abstract level. It is based on modelling approaches such as UML or SysML, and allows a description of the hardware of the system under development on an abstract level, close to what is custom in software development. Thus, development of hardware and software can proceed in a uniform and comprehensive environment; in fact, in the early stages of the development, the engineer does not need to distinguish between the two. On the formal specification level, properties such as deadlocks, inconsistencies and dynamic properties expressed in OCL can be verified [13], and flaws detected very early in the design flow, reducing costs and effort. This addresses the hardware/software divide, but also problems arising from changing environments.

C. Functional Change Management

Our approach to *functional change management* [7] is illustrated in Fig. 1. Technically, we consider all artefacts occuring in the development process and their semantics uniformly as graphs. For documents in XML or similar representation, this is the obvious document tree with links as additional edges; for the semantics, this is the called *development graph*, where nodes are the entities of the development such as specifications, modules, safety requirements, single functions, or proofs, and where edges include relations such as “satisfies”, “implements”, or “depends-on”, *e.g.* connecting a safety requirement with a function which implements the required safety functionality, and a proof verifying that.

The uniform representation with graphs allows us to formalise the semantics abstraction and change impact propagation as graph rewriting rules, and to use a graph rewriting engine (GrGen) as a common implementation basis. These rules can specify either automatic changes, or can trigger further user interaction; *e.g.* they could specify that when a safety requirement changes, the functions which implement it and the referenced proofs need be checked as well.

This approach is flexible, and has been adapted to a variety of settings, including software verification [12] or development of safety-critical systems [8]. To cover the development of cyber-physical systems, we must adapt it further to cover FSL specifications, and to cover the artefacts occuring in the electronic system level or the register transfer level (see below).

Functional change management addresses the issues of frequent changes. It makes the development process more agile, because it allows rapid feedback loops. It also addresses the issue of reuse, which is particularly relevant in system development (according to a recent study [14], in 2010 76% of all designs included at least one embedded processor, and the external IP adoption increased by 69% from 2007 to 2010).

The full integration of these three features into the design flow will change it beyond what is known and is in use today, and will result in a new design flow targeted specifically at the needs of cyber-physical systems.

IV. A New Design Flow for Cyber-Physical Systems

We illustrate the proposed design flow with a scenario illustrating the design of a (simplified) traffic light system.

A. Exploring the New Design Flow

The starting point is the description of the system in natural language. It comprises definitions such as the following

> *A traffic light consists of a light for cars, a light for pedestrians, and a request button for pedestrians. Lights are either green or red. The button counts the number of actuations. The lights for cars and pedestrians are never green at the same time.*

and various use cases such as

(1) A pedestrian pushes the button.
(2) The light for cars gets red.
(3) The light for pedestrians gets green.
(4) The light for pedestrians stays green for some period of time.
(5) The light for pedestrians gets red.
(6) The light for cars gets green.

This informal description is translated to a *formal* requirement specification of the intended system on the FSL. Analysing the phrase structure of the sentences (*e.g.* using the Stanford Parser [15]) and grouping nouns, verbs, adjectives, and adverbs into sets of cognitive synonyms (*e.g.* using WordNet [16]), we can conclude that *light* and *button* represent components of the considered system (to be represented by classes or attributes) and verbs like *push* correlate to operations.

Fig. 2 illustrates a possible result of this computer-aided transformation process. Manual interaction by the designer is necessary to define the pre- and postconditions of operations (*e.g.* *switchCarLight* or *switchPedLight*) and class invariants (*e.g.* class *TrafficLight*).

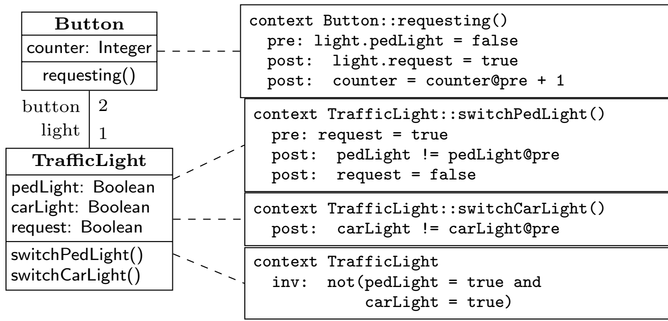

Fig. 2. UML class diagram for traffic light scenario

Based on this formal specification, the overall behaviour of the system is checked for various correctness properties (*e.g.* liveness and safety properties). Further, test cases can be derived from the informally specified use cases. In applying them, the tool detects a potential deadlock that is evident in our example. In order to reach a system state where pedestrians get a 'green' light, first *requesting* has to be invoked (assigning *request* to *True*). Due to the invariant, *switchCarLight* has to be executed next in order to set *carLight* to *False*. Finally, the call of *switchPedLight* leads to the desired system state. However, no further operations are applicable in this state since (1) the pre-conditions of *requesting* and *switchPedLight* fail and (2) the call of *switchCarLight* would lead to a system state which contradicts the invariant. Therefore, the operations of traffic light have to be adapted accordingly and the changes propagated to the other abstraction levels.

If the system is specified as intended and exhibits the required properties, the respective components are translated to the next level of the design flow, the electronic system level (ESL), where we model the system on a concrete, executable level. A typical ESL language is SystemC, a C++ class library providing hardware constructs and data types, which allows the system behaviour to be simulated at an early stage of the design flow. For the translation into the ESL, code generation methods are applied to derive appropriate SystemC specifications. As the correctness of the overall behaviour of the design has been checked at the FSL, only the ESL implementations of the individual components are left to be verified.

Finally, the ESL specification is translated to the lowest level of our design process, called register transfer level (RTL), based on which the individual hardware components are designed. On the RTL, the system is described in a hardware description language such as Verilog or VHDL.

Translating the specification of *button* of the traffic light to RTL we may realise that we have already designed a button in a previous project and we now like to reuse this IP (intellectual property, *i.e.* previous or acquired development). However, as it turns out, the behaviour of the existing button is slightly different as its internal counter is incremented by an arbitrary value provided through an additional input (instead of a fixed value 1). Hence, reusing this design of a button we have to adapt the usage of the *counter* in the traffic light system accordingly and propagate these changes back across the design flow to the higher levels, ensuring the continued correctness of the overall development. Assuming that the behaviour of the original system was verified at a higher

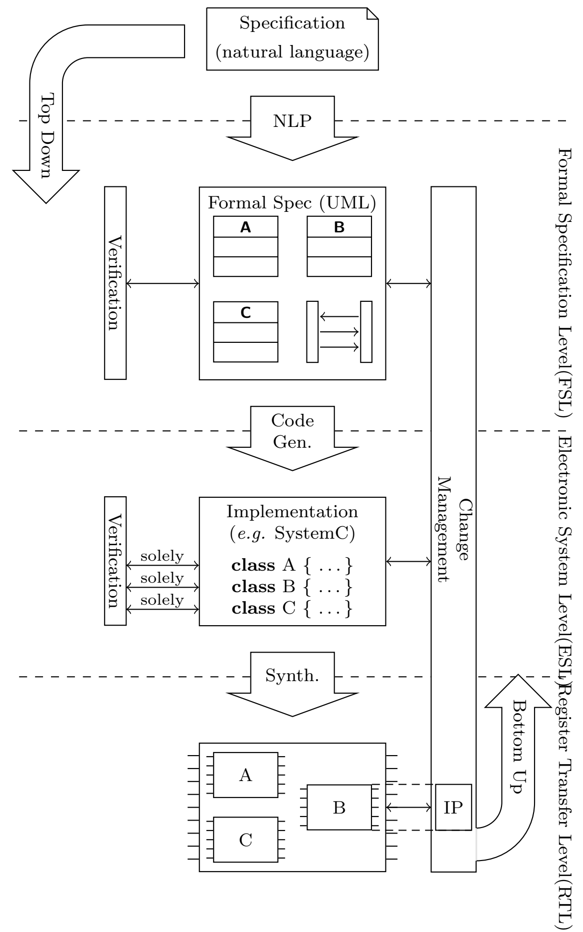

Fig. 3. Proposed design flow

abstraction level, the changes which have their origin in a lower abstraction level might invalidate the verification results at a higher level. To avoid a time-intensive complete new verification run, it would be helpful to know which proof obligations are still valid in spite of the change and which proof obligations have to be checked again. This kind of change impact analysis is handled by the functional change management.

B. The Stages of the New Design Flow

The proposed new design flow is sketched in Fig. 3. The design flow originates with specifications formulated in natural language. These are translated semi-automatically into formal specifications, using techniques as sketched above. In case of ambiguities or unresolvable specifications, the system will ask the user to provide a clarification. Verification at the FSL will allow to detect flaws early in the development process, saving time and development costs.

FSL specifications will be translated automatically into the ESL. The challenge is to translate annotations such as pre- and postconditions from the FSL into a more concrete representation at this level, and to identify at which level (FSL, ESL or even RTL) which property can be verified. The translation from ESL to RTL, finally, is routine. Further, all these transformations must maintain traceability, as we

want to be able to trace defects found in lower levels of the development — *e.g.* a timing flaw found at the RTL — back to the higher levels.

The comprehensive functional change management covers all levels of the design flow. It keeps track of proof obligations, and reports which verification tasks have to be repeated after a functional change and which properties are not affected and are still valid. This will reduce verification loops, and makes the development more agile.

The change management also covers the reuse of existing IP. When we need to integrate existing IP, this may require changes at the lower levels which impact on the higher levels; the change management lets us compute the precise effect of these changes.

V. Conclusions and Outlook

We have introduced the SPECifIC design flow, which combines natural language processing, functional change management, and the new formal specification level. It arose from our experiences in previous projects concerned with hardware or software development, and addresses some of the key challenges of developing cyber-physical systems.

The design flow will be put into place over the next two-and-a-half years during the SPECifIC project, which has started in July 2013. It will involve our industrial partners, and hence will not be a purely academic prototype.

As an example, we are currently applying our techniques to industrial benchmarks. In particular, we are addressing the automatisation of requirement formalisation. Algorithms aid to cluster requirements for an easier translation or help in detecting requirements in a larger document. From this evaluation we are pointed to more specific needs and adjust our tools accordingly.

With the aim of producing something concretely useful, the SPECifIC design flow does not address every possible problem straight away; we rather chose to address the from our point of view most potentially useful issues first, and follow up later with concerns such as the following:

- Besides the UML, it would also make sense to integrate more expressive modelling techniques such as hybrid or timed automata, and logics such as temporal or modal logic.
- The natural language processing could be extended to cover these formalisms as well. In fact, in our view it is important to keep the natural language open, and not restrict ourselves to a particular subset, as this allows us to add in more expressiveness into the language at a later point.
- We also think of replacing conventional design tools which are in use today by collaborative tools to enable a continuous connection between all stakeholders in the design flow. An initial prototype [17] can be used for this purpose and serve as a central tool inside the SPECifIC design flow.
- Another issue is the long-term autonomy of CPS, which may operate for a substantial or even indefinite amount of time without the possibility to update or amend them. This requires self-modification or even self-healing capabilities, which it is unclear how to describe formally.

References

[1] R. Drechsler, B. Becker, and S. Ruppertz, "The K*BMD: A verification data structure," *IEEE Design & Test of Computers*, vol. 14, no. 2, pp. 51–59, 1997.

[2] R. Drechsler and D. Sieling, "Binary decision diagrams in theory and practice," *STTT*, vol. 3, no. 2, pp. 112–136, 2001.

[3] R. Drechsler, S. Eggersglüß, G. Fey, A. Glowatz, F. Hapke, J. Schlöffel, and D. Tille, "On acceleration of SAT-based ATPG for industrial designs," *IEEE Trans. on CAD of Integrated Circuits and Systems*, vol. 27, no. 7, pp. 1329–1333, 2008.

[4] S. Eggersglüß and R. Drechsler, "A highly fault-efficient SAT-based ATPG flow," *IEEE Design & Test of Computers*, vol. 29, no. 4, pp. 63–70, 2012.

[5] A. Sulflow, U. Kuhne, G. Fey, D. Grosse, and R. Drechsler, "WoLFram – a word level framework for formal verification," in *RSP '09. IEEE/IFIP International Symposium on Rapid System Prototyping*, june 2009, pp. 11– 17.

[6] D. Hutter, "Management of Change in Structured Verification," in *Proceedings 15th IEEE International Conference on Automated Software Engineering*, ser. ASE, no. 2000. IEEE Computer Society, 2000, pp. 23–34.

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

[8] S. Autexier, D. Dietrich, D. Hutter, C. Lüth, and C. Maeder, "SmartTies - management of safety-critical developments," in *Proceedings 5th International Symposium On Leveraging Applications of Formal Methods, Verification and Validation (ISoLa'12)*, ser. LNCS, T. Margaria and B. Steffen, Eds., vol. 7609. Springer, 2012, pp. 238– 252.

[9] H. Täubig, U. Frese, C. Hertzberg, C. Lüth, S. Mohr, E. Vorobev, and D. Walter, "Guaranteeing functional safety: design for provability and computer-aided verification," *Autonomous Robots*, vol. 32, no. 3, pp. 303–331, April 2012.

[10] D. Walter, H. Täubig, and C. Lüth, "Experiences in applying formal verification in robotics," in *SafeComp 2010 — 29th International Conference on Computer Safety, Reliability and Security*, ser. LNCS, vol. 6351. Springer, 2010, pp. 347– 360.

[11] C. Lüth and D. Walter, "Certifiable specification and verification of C programs," in *Formal Methods (FM 2009)*, ser. LNCS, A. Cavalcanti and D. Dams, Eds., vol. 5350. Springer, 2009, pp. 419– 434.

[12] S. Autexier and C. Lüth, "Adding change impact analysis to the formal verification of C programs," in *iFM 2010: Integrated Formal Methods - 8th International Conference*, ser. Lecture Notes in Computer Science, D. Méry and S. Merz, Eds., vol. 6396. Springer, 2010, pp. 59– 73.

[13] M. Soeken, R. Wille, and R. Drechsler, "Verifying dynamic aspects of UML models," in *Design, Automation and Test in Europe*, 2011, pp. 1077–1082.

[14] Wilson Research Group and Mentor Graphics, "2010-2011 Functional Verification Study," 2011.

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

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

[17] C. Lüth and M. Ring, "A web interface for Isabelle: The next generation," in *Conferences on Intelligent Computer Mathematics CICM 2013*, ser. LNAI, J. Carette, Ed., vol. 7961. Springer, 2013, pp. 326–329.