CONTINUOUS ENGINEERING OF EMBEDDED SYSTEMS

Bernhard Steffen [1], Tiziana Margaria[2]
[1] *Chair of Programming Systems, Universität Dortmund (Germany)*
[2] *Chair of Service and Software Engineering, Universität Potsdam (Germany)*

Abstract We investigate the late phases of the embedded systems' life cycles, in particular the treatment of change requests, the integration of legacy components, and the problem of emerging platforms. We propose to tackle these issues in a model-driven design paradigm, on the behavioral models, and to employ techniques from automata theory, model checking and automata learning. The main practical impact of our approach is its support of the systematic *completion* and *update* of user/customer requirements, which by their nature are quite partial and concentrate on the most prominent scenarios. Our technique generalizes these typical requirement skeletons by extrapolation and it indicates via automatically generated traces where the requirement specification is too loose and additional information is required. This works in the initial phases of system development, but also in case of change requests, where our technique hints at possible problems with their realization (feature interactions), and helps to keep the requirement model in synchrony along the chain of new releases.

1. MOTIVATION

The bulk of research concerning embedded systems focusses of the early stages of the systems' life cycles. Today's systems require unacceptable efforts just for deployment, typically caused by incompatibilities, feature interactions, and by the sometimes catastrophic behavior of component upgrades, which no longer behave as expected. This is particularly true for embedded systems, with the consequence that some components' lifetimes are 'artificially' prolonged far beyond a technological justification, for fear of problems once they are substituted or eliminated.

The time after the first deployment causes the majority of the overall costs of the system, but is hardly addressed, as is the integration of legacy components. It is a major research challenge to provide systematic and

Please use the following format when citing this chapter:

Steffen, B., Margaria, T., 2006, in IFIP International Federation for Information Processing, Volume 225, From Model-Driven Design to Resource Management for Distributed Embedded Systems, eds. B. Kleinjohann, Kleinjohann L., Machado R., Pereira C., Thiagarajan P.S., (Boston: Springer), pp. 45–54.

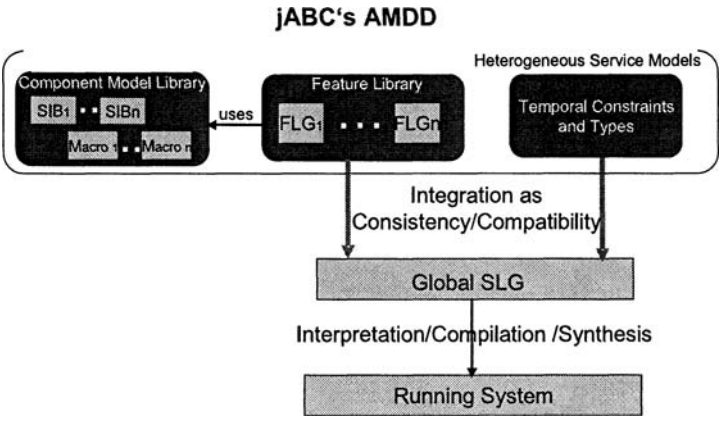

Figure 1. The AMDD Process in the jABC

consistent support to the later phases of the life cycle as well as system construction with legacy components. Prerequisite in this direction is the development of modelling levels that capture these aspects.
Here we investigate this situation under three perspectives, that treat

1 *changing requests*: in particular, the consistent integration of new requirements.

2 *legacy components*: how can we increase our confidence when dealing with components lacking specification.

3 *emerging platforms*: how can we separate the two issues above from the technological details required for effective technology mapping.

Starting point for our analysis is an *aggressive* version of model-driven development (AMDD) [14], which moves most of the recurring problems of compatibility and consistency of system design, implementation, and evolution from the coding and integration levels to the modelling level (Fig. 1). Being a paradigm of *system design*, it inherently leaves a high degree of freedom in the design of adequate settings. We propose to treat the first two issues at the level of *behavioral models*, and use automata theory, model checking, and automata learning to consistently deal with looseness of requirements, uncertainties of legacy components, and the consequences of change requests. The third point coincides with the technology mapping issue discussed in the context of AMDD [14].

The main practical impact of the technique proposed in this paper is its ability to support the systematic *completion* and *update* of user/customer requirements, which by their nature are typically very

partial and concentrate on the most prominent scenarios. Our technique generalizes these typical requirement skeletons by learning-based extrapolation, and it indicates via automatically generated traces where the requirement specification is too loose and additional information is required. This technique can also be used for the construction of behavioral models for third party/legacy components [5; 12].

This works in the initial phases of system development, but also with change requests, where our technique hints at possible problems with their realization (feature interactions), and helps to semi-automatically keep the requirement model in synchrony along the chain of new releases.

In the following, Sect. 2 briefly sketches our tool landscape, based on the jABC Modelling Framework, and Sect. 3 the essential features of automata learning. Then Sect. 4 presents our method of learning-based long-term requirement engineering: the principle, the simple monotonic case of requirement completion, and the non-monotonic case of requirement updates. Finally Sect. 5 gives our conclusions and perspectives.

2. BASICS CONCEPTS OF THE jABC

jABC [6] is a Java-based framework for service development along the AMDD paradigm [14]. Its central model structure are hierarchical, flow chart-like graphs called Service Logic Graphs (SLGs) [13]. They model the application behavior in terms of the intended process flows, based on coarse granular building blocks called SIBs (Service-Independent Building blocks) which are to be understood directly by the application experts – independently of the structure of the underlying code, which in our case is typically written in Java/C/C++. The component models (single SIBs or hierarchical subservices), the feature-based service models called Feature Logic Graphs (FLGs), and the Global SLGs modelling applications are all hierarchical SLGs. Additionally, the jABC supports several *model specification* styles, including

1 two modal logics, mu-calculus and monadic second order logic on strings [19], to abstractly and loosely characterize valid behaviors of finite and parametric systems, resp., which come with plugins for the graphical, pattern-based definition of constraints [7],

2 a classification scheme for building blocks and types, useful e.g. for service discovery in distributed service environments [11], and

3 model-level and source code-level analysis and consistency verification mechanisms based on these specifications [2].

In this sense, the jABC is an instance of *actor-based* modelling environments according to [3].

Predecessors of jABC have been used since 1995 to design, among others, industrial telecommunication services [15], Web-based distributed decision support systems [8], and test automation environments for Computer-Telephony integrated systems [5], and most recently to equip the Ricoh AFICIO printer series with a flexible process management.

jABC allows users to develop services and applications by composing reusable building-blocks into (flow-)graph structures. An extensible set of plugins provides additional functionalities to adequately support all the activities needed along the development lifecycle, like e.g. animation, rapid prototyping, formal verification, debugging, code generation, monitoring, and evolution. This process does not substitute but rather enhance other modelling practices like the UML-based RUP 17, which is in fact used in our process to design the single components.

jABC offers a number of advantages that play a particular role when integrating off-the-shelf, possibly remote functionalities.

- **Agility**. We expect requirements, models, and artifacts to change over time, therefore the process supports evolution as a normal process phase by various means, like, e.g., model checking, monitoring-based consistency checking, and requirement completion/update.
- **Consistency**. The same modelling paradigm underlies the whole process, from the very first steps of prototyping up to the final execution, guaranteeing traceability semantic consistency.
- **Verification**. With techniques like model checking and local checks we support the user to consistently modify his model. The basic idea is to provide automatic checking mechanisms for previously defined local or global properties that the model must satisfy.
- **Service orientation**. Legacy or external features, applications, or services can be easily integrated into a model by wrapping the functionality into building blocks to be used inside the models.
- **Executability**. The models can be executed in various modes: executions can be as abstract as guided documentation browsing and as complex as the concrete run of the final implementation.

Several applications have shown how these properties are exploited in different application areas, like e.g. [9; 4; 8; 10].

3. AUTOMATA LEARNING

Machine learning deals in general with the problem how to automatically generate a system's description. Besides the synthesis of static soft-

and hardware properties, in particular invariants, the field of *automata learning* is of particular interest for soft- and hardware engineering.

Automata learning tries to construct a deterministic finite automaton (see below) that matches the behavior of a given target automaton on the basis of observations of the target automaton and perhaps some further information on its internal structure. The interested reader may refer to [18] for details, here we only summarize the basic aspects of our realization, which is based on Angluin's learning algorithm L^*[1].

L^*, which is an *active* learning algorithm, learns finite automata by *actively* posing *membership* queries and *equivalence* queries to the target automaton in order to extract behavioral information, and refining successively an own *hypothesis automaton* (HA) based on the answers. Membership queries test whether a string (a potential run) is contained in the target automaton's language (its set of runs), and equivalence queries compare the HA with the target automaton for language equivalence, in order to determine whether the learning procedure has (already) successfully completed and the experimentation can stop.

In its basic form, L^* starts with the one state HA that treats all words over the considered alphabet (of elementary observations) alike, and refines it on the basis of query results iterating two steps. Here, the dual way of how L* characterizes (and distinguishes) states is central:

- from *below*, by words reaching them. This characterization is too fine, as different words may well lead to the same state.
- from *above*, by their future behavior wrt. a dynamically increasing set of words. These future behaviors are essentially bitvectors: a '1' means that the corresponding word of the set is guaranteed to lead to an accepting state and a '0' captures the complement. This characterization is typically too coarse, as the considered sets of words are typically rather small.

The second characterization directly defines the HAs: each occurring bitvector corresponds to one state.

The initial HA is characterized by the outcome of the membership query for the empty observation. Thus it accepts any word if the empty word is in the language, and no word otherwise. Next, the learning procedure (1) iteratively establishes local consistency, after which it (2) checks for global consistency.

Local Consistency. This is an automatic *model completion* loop, that iterates two phases: (a) checking whether the constructed HA is *closed* under the one-step transitions, i.e., each transition from each state

of the HA ends in a well defined state of this very automaton. And (b) checking *consistency* according to the bitvectors characterizing the future behavior as explained above, i.e., whether all reaching words with an identical characterization from above have the same one step transitions. If this fails, a distinguishing transition is taken as an additional *distinguishing future* that resolves the inconsistency, splitting the state.

Global Equivalence. After local consistency, an equivalence query checks whether the language of the HA coincides with that of the target automaton. If this is true, the learning procedure successfully terminates. Otherwise the equivalence query returns a counterexample, i.e., a word which distinguishes the hypothesis and the target automaton. This counterexample gives rise to a new cycle of modifying the HA and starting the next iteration.

4. LEARNING-BASED REQUIREMENTS MANAGEMENT

Key towards continuous engineering is the treatment of new/changing requirements. Our learning-based approach to requirement management is organized to capture three dimensional requirement evolution:

- Use cases: individual 'runs' of the intended system.
- Temporal properties: global constraints that capture safety and liveness properties of the considered system.
- Structure requirements: system properties like symmetry between technical components, determinacy of individual behaviors,

Change request may concern all three dimensions, although individual change requests typically belong to one. This means in particular that the other dimensions remain unchanged, which drastically enhances the stability of upgrading procedures.

It is the central data structure of active automata learning, the *observation table*, which, slightly enhanced, enables the incremental and evolutionary model construction. It does not only comprise the actual HA, but also the concrete evidence which lead to its construction. Thus it allows us to distinguish between the model structure based on concrete observations and the model structure which arose in the course of extrapolation: the HAs are the state-minimal automata consistent with the actual observations. As such, they are neither an under- nor an over-approximation: they may as well allow (extrapolated) behavior, which the considered system will never engage in, as also refuse behavior the considered system is capable of.

The enhanced observation tables are adequate as a means for change management at the requirement level: they indicate how to cover new or changing requirements without violating the primary model structure. The basic intuition behind our approach is the following technique for completing and changing requirement specifications.

4.1 COMPLETING REQUIREMENT SPECIFICATIONS

Requirement specifications in terms of individual traces are by their nature very partial and represent only the most prominent situations. This partiality is one of the major problems in requirement engineering: it often causes errors in the system design that are difficult to fix. Thus techniques for systematically completing such partial requirement specifications are of major practical importance.

We propose a method for requirements completion based on automatic (active) automata learning that in essence

- *initializes* the learning algorithm with the set of traces constituting the requirement specifications, and
- constructs a *consistent behavioral model* by establishing the local consistency introduced in the previous section.

This way, we build a finite state behavioral model which *extrapolates* the given requirement specification: it comprises *all* 'positive' traces of the specification, and rejects all forbidden traces. All the other potential traces are consider as 'don't cares', in order to construct a corresponding state minimal HA. In particular, although the learning procedure by its nature will only investigate finitely many traces, the constructed HA will typically accept infinitely many traces, since the extrapolation process introduces loops.

For this method to work, a number of membership queries need to be answered. Both, establishing closure of the model, as well as establishing the consistency of the abstraction of reaching words into states (i.e., of the characterization from above introduced in the previous section) can only be achieved on the basis of additional information about the intended/unknown system. This is not unexpected, rather even desired (at least to some extent): the posed membership queries directly hint at the places where the given requirement specification is partial. On the other hand, it is not practical: the number of such membership queries is the major bottleneck of active learning, even when its generation is fully automated. The execution of membership queries is interactive, and the effort unacceptable.

We observed that the number of membership queries can be drastically reduced on the basis of orthogonally given expert knowledge about the intended/unknown system. We could show that already the following three very general structural criteria, *prefix closure*, *independence* of actions and *symmetry*, were sufficient to reduce the number of membership queries by several orders of magnitude [5; 12].

This idea, originally designed for the optimization of the treatment of 'use case'-based requirement completion, allows us to also capture the other two dimensions of requirement specification:

Temporal properties:. global constraints that capture safety and liveness properties of the system. Besides typical example runs, application experts are also able to formulate many necessary safety conditions, be it on the basis of required protocols, or the exclusion of catastrophic states. Adding these safety requirements in terms of temporal logics to our specification can automatically answer a huge number of membership and equivalence queries, via model checking. This way, these properties are automatically taken care of during model construction.

Structure requirements:. concern the shape of the overall system. The nature of the system to be learned often leads to structural constraints, like symmetry between technical components, or determinacy of individual behaviors. These constraints can be automatically taken care of by corresponding operations on automata, which add this structure to the observed behavioral skeleton.

4.2 CHANGING REQUIREMENT SPECIFICATIONS

So far our learning scenario was *monotonic*: observations made once are guaranteed to remain valid. Thus it might only happen that

- new observations revive the learning process, and/or
- assumed temporal of structural properties turn out to be false

forcing us to revise the hypothesis model. In both cases we can incrementally deal with membership queries – the major bottleneck – and in fact the classical observation table supports this incremental treatment.

In contrast to requirement completion, requirement update is non-monotonic, and may have a destructive effect on the observations made. It may force us to discharge some results of previously answered membership queries. There are two extreme approaches (and of course numerous compromises) to do so:

- start the learning procedure from scratch, which is of course not incremental, but inherits all the nice properties of automata learning, like 'the HA is a state-minimal representative of all automata consistent with the made observations'.

- continue the learning process as if it were monotonic, only question previous query results in case there is a conflict, and in case of conflict give the new observation precedence.

  This approach is incremental, but unfortunately conflict resolution is not as easy as it might seem: a new trace may well be in conflict to quite a number of traces. Perhaps even worse, an old (no longer valid) observation may not be conflicting with the new observation but make the hypothesis model 'explode'. We are currently investigating various strategies to overcome these problems.

On the other hand, if we maintain knowledge about which queries were answered by model checking or the assumption of structural properties (our enhancement of the observation table), one can rather comfortably deal with the change of these kinds of requirements.

5. CONCLUSIONS AND PERSPECTIVES

We have presented an approach to support the systematic *completion* and *update* of user/customer requirements along a system's life cycle. This method, mainly based on automata learning, elegantly complements our behavioral model construction for legacy systems, and provides a powerful support for the late phases of the systems' life cycles.

Currently, we are investigating how to efficiently deal with non-monotonic updates. Due to the three dimensional structure of the considered space of requirement specification, we do not expect a unique answer. In fact, we believe that, in practice, one will very much depend on additional information about which kind of changes have been made, in order to derive an efficient strategy for learning-based model update.

REFERENCES

[1] D. Angluin. Learning regular sets from queries and counterexamples. *Information and Computation*, 2(75):87–106, 1987.

[2] A.L. Lamprecht, T. Margaria, B.Steffen: *Data-Flow Analysis as Model Checking within the jABC*, Proc. CC'06, 15th Int. Conf. on Com-piler Construction, Vienna (A), March 2006, LNCS, 3923, Springer Verlag, pp. 101-104.

[3] E. Lee, S. Neuendorffer, M. Wirthlin. *Actor-oriented design of embedded hardware and software systems* Journal of circuits, systems, and computers. 2002.

[4] M. Hörmann, T. Margaria, T. Mender, R. Nagel, M. Schuster, B. Steffen, H. Trinh: *The Jabc Approach To Collaborative Development of Embedded Applica-*

tions, CCE'06, Worksh. on Challenges In Collaborative Engineering (Industry day), Prag, April 2006.

[5] H. Hungar, T. Margaria, B. Steffen: *Test-Based Model Generation for Legacy Systems*, Proc. IEEE ITC'03, Charlotte, 2003, IEEE CS Press, pp.971–980.

[6] jABC Webseite: www.jabc.de

[7] S. Jörges, T. Margaria, B. Steffen: *FormulaBuilder: A Tool for Graph-based Modelling and Generation of Formulae*, Proc. ICSE 2006, 28th ACM-IEEE Int. Conf. on software Engineering, Shanghai (CHN), May 2006, to appear.

[8] M. Karusseit, T. Margaria: *Feature-based Modelling of a Complex, Online-Reconfigurable Decision Support Service*, WWV'05, 1st Int. Worksh. Automated Specification and Verification of Web Sites, Valencia, March 2005, ENTCS 1132.

[9] C. Kubczak, R. Nagel, T. Margaria, B. Steffen: *The jABC Approach to Mediation and Choreography*, Semantic Web Services Challenge 2006, Phase I Workshop, DERI, Stanford University, Palo Alto, March 2006.

[10] T. Margaria, C. Kubczak, M. Njoku, B. Steffen: *Model-based Design of Distributed Collaborative Bioinformatics Processes in the jABC*, IEEE ICECCS 2006, Stanford, Aug. 2006, to appear.

[11] T. Margaria, R. Nagel, B. Steffen: *Remote Integration and Coordination of Verification Tools in JETI*, Proc. IEEE ECBS 2005, April 2005, Greenbelt (USA), IEEE CS Press, pp. 431–436.

[12] T. Margaria, H. Raffelt, B. Steffen: *Knowledge-based relevance filtering for efficient system-level test-based model generation*, Innov. in System and Software Engineering - a NASA Journal, Vol. 1(2), pp.147-156, Springer Verl., Sept. 2005.

[13] T. Margaria, B. Steffen: *Lightweight Coarse-grained Coordination: A Scalable System-Level Approach*, STTT, Int. Journal on Software Tools for Technology Transfer, Special section on Formal Methods for Industrial Critical Systems, Vol.5, N.2-3, 2004, Springer Verlag, pp.107-123.

[14] T. Margaria, B. Steffen: *Aggressive Model-Driven Development: Synthesizing Systems from Models viewed as Constraints*, Workshop on Software Engineering for Embedded Systems: From Requirements to Implementation, The Monterey Workshop Series, Chicago, September 2003.

[15] T. Margaria, B. Steffen, M. Reitenspieß: *Service-Oriented Design: The Roots*, ICSOC 2005: 3rd ACM SIGSOFT/SIGWEB Int. Conf. on Service-Oriented Computing, Amsterdam, Dec. 2005, LNCS 3826, pp. 450-464, Springer V..

[16] H. Raffelt, B. Steffen, T. Berg: *LearnLib: A Library for Automata Learning and Experimentation*, Proc. FMICS 2005, 10th ACM Workshop on Formal Methods for Industrial Critical Systems, Lisbon, Sept. 2005, ACM Press, pp.62–71.

[17] Rational Unified Process. http://www-306.ibm.com/software/awdtools/rup/

[18] B. Steffen and H. Hungar, Behavior-based model construction. In S. Mukhopadhyay and L. Zuck, editors, *Proc. 4th Int. Conf. on Verification, Model Checking and Abstract Interpretation*, LNCS 2575, Springer 2003, pp.5–19.

[19] C. Topnik, E. Wilhelm, T. Margaria, B. Steffen: *jMosel: A Stand-Alone Tool and jABC Plugin for M2L(Str)*, Proc. SPIN'06, 13th Int. SPIN Works. on Model Checking of Software, Vienna, April 2006, LNCS 3925, Springer V., pp.293-298.