Language Engineering as an Enabler for Incrementally Defined Formal Analyses

Daniel Ratiu*, Markus Voelter†, Bernhard Schaetz*, Bernd Kolb‡
* *ForTISS, Munich, Germany, {ratiu | schaetz}@fortiss.org*
† *independent / itemis, Stuttgart, Germany, voelter@acm.org*
‡ *itemis, Stuttgart, Germany, kolb@itemis.de*

***Abstract*—There is a big semantic gap between today's general purpose programming languages on the one hand and the input languages of formal verification tools on the other hand. This makes integrating formal analyses into the daily development practice artificially complex. In this paper we advocate that the use of language engineering techniques can substantially improve this situation along three dimensions. First, more abstract and thus more analyzable domain specific languages can be defined, avoiding the need for abstraction recovery from programs written in general purpose languages. Second, restrictions on the use of existing languages can be imposed and thereby more analyzable code can be obtained and analyses can be incrementally defined. Third, by expressing verification conditions and the verification results at the domain level, they are easier to define and the results of analyses are easier to interpret by end users. We exemplify our approach with three domain specific language fragments integrated into the C programming language, together with a set of analyses: completeness and consistency of decision tables, model-checking-based analyses for a dialect of state machines and consistency of feature models. The examples are based on the mbeddr stack, an extensible C language and IDE for embedded software development.**

I. Introduction

A. Problem Context

Formal verification techniques have great potential and have reached levels of scalability that makes them suitable for real-world systems. However, they are not used by mainstream developers, even though many could benefit. There are several reasons for this situation, one of them being the perception by practitioners that formal methods are only for experts, and require the use of sophisticated tools and languages. Another more technical reason is nicely described in [5]. We cite:

The transfer of [formal verification techniques] from research to practice has been much slower for software. One reason for this is the model construction problem: the semantic gap between the artifacts produced by software developers and those accepted by current verification tools. Most development is done with general-purpose programming languages (e.g., C, C++, Java, Ada), but most verification tools accept specification languages designed for the simplicity of their semantics (e.g., process algebras, state machines). In order to use a verification tool on a real program, the developer must extract an abstract mathematical model of the program's salient properties and specify this model in the input language of the verification tool. This process is both error-prone and time-consuming.

Although this has been written in the year 2000, the statements made in this paragraph are fundamentally still valid. The conceptual gap between the language in which programs are expressed and the language of the formal analysis tool needs to be overcome.

One way to address this problem is to generate the input to verification tools from higher-level, more "friendly" descriptions of the functionality of the system. This approach is used by modeling tools such as Simulink or Statemate. However, this approach has problems as well. One problem is the integration between those parts of the system expressed in higher-level models and the rest of the system that is typically still written in a general purpose language (GPL). Another problem is the limited support for incrementally adding formal verification to parts of the system as the need arises: the respective part has to be removed from the GPL code and redescribed completely in the modeling tool. The situation gets even worse if different parts of the system have to be verified and/or modeled in different ways, with different tools.

In essence, developers have to make a back-and-white decision: either use a GPL and thereby lose much of the verifiability of domain abstractions, or use specialized tools, and suffer from the lack of integration with existing code bases. We claim that this decision can be made smoother and more nuanced by using incremental language extensions on the basis of a GPL, C in our case.

B. The mbeddr Approach

In mbeddr we investigate a different approach that can be considered a middle ground between the two approaches described above (Figure 1). mbeddr uses language engineering in two ways: (a) to incrementally add

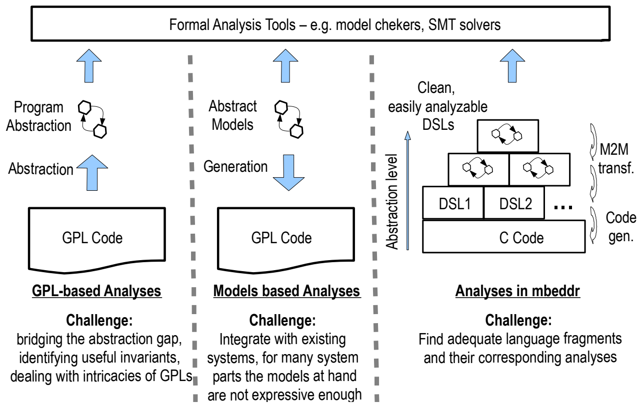

Figure 1. Analyses approaches at a glance

abstractions to an existing base language and thereby make the programs easier to verify, or, (b) to restrict the language to subsets that are easier to analyze.

As a consequence, an end user can make a decision whether a program fragment should be verifiable or not: if it should be verifiable, it must be expressed with language abstractions that facilitate verification. However, the end user does *not* have to change the tool and remodel everything: he simply includes the additional language module and iteratively refactors his code to use the more suitable abstractions. The transition is seamless.

This incremental approach also works for the language and verification *developer* as opposed to the end user. If a new verification approach should be supported, the existing base language (C in our case) can be extended with the necessary additional language concepts. The extensions live in a separate module and require no changes to the base language. The developer then creates a transformation from the new abstractions back to C (for implementation) and to the input language of the verification tool. He also has to define how the output of the verifier relates back to the abstractions.

We believe this is a promising approach because it supports incremental integration of formal analyses into (existing) programs and it does not require the end user to leave the code-centric development environment.

C. *Structure of the Paper*

In Section II we present an overview over the mbeddr technology stack that represents the basis for our approach to define languages that are easier to analyze. We then present examples of domain specific extensions on top of C and the analyses performed based on them in Section III. In Section IV we sketch a methodology for performing formal analyses, and Section V discusses different variability points of our approach. We conclude the paper by discussing related work in Section VI and provide an outlook on future work in Section VII.

II. The mbeddr Technology Stack

mbeddr is an open source project (hosted at *http://mbeddr.com*) that enables embedded software development based on an extensible version of the C programming language. It supports the incremental, modular domain-specific extension of C. In addition to language *extension*, the approach also supports language restriction, in order to create subsets of existing languages. In this section we describe the stack in more detail. Figure 2 shows an overview.

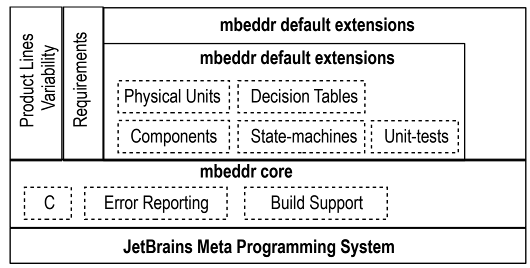

Figure 2. The mbeddr technology stack rests on the MPS language workbench. The first language layer contains an extensible version of the C programming language plus special support for logging/error reporting and build system integration. On top of that, mbeddr comes with a set of default C extensions (components, state machines, unit tests and physical units) plus cross-cutting support for requirements, traceability and product line variability.

A. *JetBrains Meta-Programming System*

As the foundation, the mbeddr stack is built on top of the JetBrains Meta Programming System. MPS is a language workbench, a tool that supports the definition, composition and use of languages. MPS supports the definition of abstract syntax, concrete syntax, type systems, transformations and generators as well as advanced IDE features such as refactorings, quick fixes and debuggers.

What distinguishes MPS from other, similar tools is that it uses a projectional editor. This means that, although a concrete syntax may look textual, it is in fact not text. In a projectional editor, a user's editing actions lead *directly* to changes in the abstract syntax tree. No grammar or parser is involved. Projection rules render a concrete syntax from the abstract syntax tree. As a consequence, MPS can work with non-textual notations such as tables, and it also supports unconstrained language composition and extension — no parser ambiguities can ever result from combining languages.

This ability to combine arbitrary languages is what we exploit in mbeddr. We use mainly language *extension*, i.e. additional languages are extensions of existing ones. The semantics of a language extension E that extends a base language B, is given by translating E back to B.

B. *An Extensible C*

The next layer in mbeddr is an implementation of the C programming language in MPS. As a consequence of how

MPS works, our C implementation is easily extensible. The implementation of C is faithful to the C99 standard, with a couple of changes that have been chosen mainly to make the C core more easily analyzable. For example, we have introduced a native *boolean* type, we force users to use the C99 integral types (*int8_t*, etc.) and we have banned the preprocessor and replaced it with first class support for its major uses such such as constants, macros and product line variability.

C. Default Extensions

While a particular domain always requires particular abstractions (realized in mbeddr with language extensions), there are many extensions that are relevant to a large subset of embedded systems. These have been implemented in a library of reusable language modules. Exploiting MPS' facilities, this means that a user, as he writes a program, can decide which language extensions he needs and import them into his program. The following extensions are available:

- *Decision tables* extend conditional expressions with a tabular representation for nested *if*-statements.
- *Test cases* provide first class support for test driven development.
- *Interfaces* with pre- and postconditions support the *specification* of functionality. Components which provide and require interfaces through ports enable modular *implementation* of interface functionality. Stubs and mocks support testing.
- *State machines* with in and out events that can be bound to C functions
- *Data types with physical units* and an extended type checker support first-class use of physical quantities.

D. Process Support

The mbeddr stack provides support for two important aspects of software engineering: requirements traceability and product line variability. Both are implemented in a generic way that makes them reusable with any mbeddr-based language.

- *Requirements traces* are annotations on program elements that contain a typed relation to a requirement. This way, a program element can be made to express for example an *implements* or a *tests* relationship with one or more requirements.
- *Feature models* support the expression of product line variability. Presence conditions can be attached to any program element to express a dependency on a particular combination of features. The program can be edited in product line mode, with the optional parts in the code annotated with the presence conditions, or it may be edited as a specific variant with those parts of the program not in the variant removed.

E. Integration of Verification Tools

mbeddr provides an integration with two verification tools. The NuSMV model checker [3] is used for model checking state machines (discussed in Section III-B). The Yices SMT solver [1] is used for checking the consistency and completeness of decision tables (Section III-A) and to verify the absence of conflicts in feature models as well as the compliance of defined configurations to the feature models (Section III-C).

III. Extensions and Analyses on Top of C

Each of the next subsection describe a language extension and a set of analyses that we implemented for the extension. In each subsections we first present the extension, then we present conceptually the analyses that are interesting for this fragment, and finally look at the implementation of these analyses.

A. Consistency and Completeness for Decision Tables

Decision tables exploit MPS' projectional editor in order to represent two-level nested *if* statements as a table. Figure 3 shows an example. Decision tables [9] let users describe different actions that can be taken for different combinations of input conditions. The rationale for tabular expressions is to let developers define the conditions more easily and to allow reviewers to directly gain an overview of varied sets of input conditions. Decision tables are translated into C essentially as an *if/else if/else* for the column headers, and nested in each branch, an *if/else if/else* for the row headers.

```
enum mode { MANUAL; AUTO; FAIL; }

mode nextMode(mode mode, int8_t speed) {
  return mode, FAIL |            | mode == MANUAL | mode == AUTO | ;
                    | speed < 30 | MANUAL         | AUTO         |
                    | speed > 30 | MANUAL         | MANUAL       |
} nextMode (function)
```

Figure 3. An example decision table

Analyzing Decision Tables: For a two-dimensional decision table, there are two obvious possible analyses:

- Completeness: requires that every behavior of the system is explicitly modeled and no case is omitted: this enforces explicitly listing all the possible combinations of the input conditions in the table.
- Consistency: check whether there are input conditions overlap, meaning that several cases are applicable for a single input value (non-determinism).

As long as the language used for expressing the decisions is kept "simple" (i. e.,logical and linear arithmetic expressions), the above analyses can be reduced to simple SMT problems. For example, given a table with n rows (r_i) and m columns (c_j), we can check its completeness by checking the satisfiability of the following formula (if satisfiable, then the table is incomplete).

$$\neg \bigvee_{i,j=1}^{n,m} (r_i \wedge c_j)$$

A very useful feature of SMT solvers is the generation of evidence for satisfiable formula. This evidence is useful to the user to understand the cases he missed while defining the table.

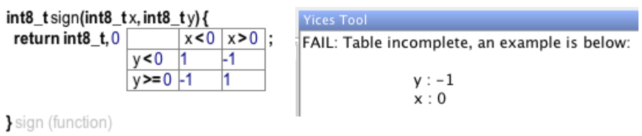

Figure 4. Checking the completeness of decision tables

Figure 4 shows an example decision table with $r_i = \{y < 0, y \geq 0\}$ and $c_j = \{x < 0, x > 0\}$. On the right hand side of this figure is given the evidence presented to the mbeddr user about the missed cases.

Similarily, the consistency of decision tables can be expressed as checking whether the following conjunctions are satisfiable. If they are satisfiable, then an inconsistency was found.

$$\forall i, k = 1..n, j, l = 1..m :$$
$$i \neq k \wedge j \neq l \Rightarrow r_i \wedge c_j \wedge r_k \wedge c_l$$

Restricting Decision Tables: Decision tables in mbeddr generally allow arbitrary conditions. These may contain function calls, or not be in the subset of linear arithmetics, which makes them not analyzable with Yices. If a user wants a decision table to be verifiable, he must restrict the expressions to this analyzable subset, essentially linear expressions. Note that the IDE reports an error in case a decision table that is marked as *verifiable* contains expressions that are not in this subset (Figure 5). This always keeps the user informed as to whether his code is verifiable or not.

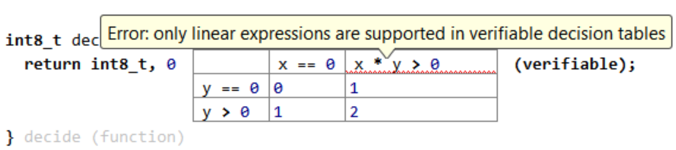

Figure 5. Defining the analyzable fragment of decision tables

B. Model-checking Statemachines

State machines are top level concepts, i.e. they reside at the same level in C programs as function or *struct*s. State machines act as types and can be instantiated. Figure 6 shows an example, where *c1* and *c2* are instances of the same state machine *Counter*. The *trigger* statement can be used to feed events into state machine instances.

State machines have in-events, out-events, states and transitions as well as local variables. Each transition is

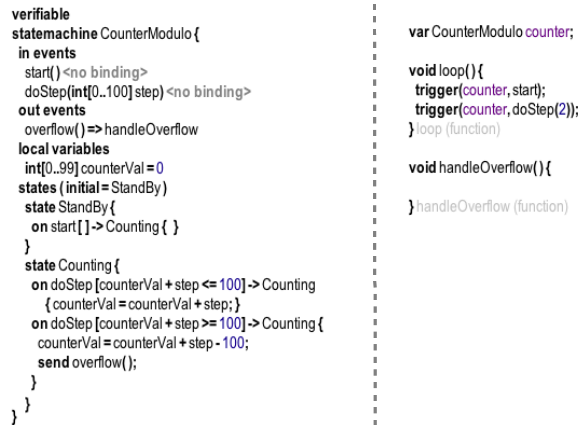

Figure 6. A state machine embedded in a C program; it resides alongside global variables and functions.

triggered by an in-event. Transitions also have guard expressions that have to be *true* in order for the transition to fire if its triggering event arrives into the state machine. The guard can refer to state machine local variables as well as to in-event parameters. A state has entry and exit actions, and transitions have transition actions. As part of actions, local variables can be assigned and out-events can be fired. As a way of interacting with the remaining C program, an out-event can be bound to a C function call (as illustrated in Figure 6).

Analyzing State Machines: Model-checking based analyses are the most suitable for the state machine language. There are numerous works (e. g.,[4]) about model-checking different dialects of state machines; mbeddr support two kinds of analyses:

- *Default analyses* are checked automatically for every state machine. These uncover typical bugs that can occur when working with state-machines: unreachable states, transitions that cannot be ever fired ("dead-code"), sets of nondeterministic transitions, and over-/underflow detection for integer variables.
- *User-defined analyses* are defined by users specifically for a given state machine. In order to address the expectations of our users (typically not experts in formal verification), we support specifications expressed with the well-known set of specification patterns[1] described in [6].

In part a) of Figure 7 we present an example of a state machine and the analyses that are run. Only one user-defined verification condition is specified (bottom-left). This condition is an example of the temporal logics verification pattern "Response - After": after the state machine is in state *Counting*, if the *Stop* event occurs, then the state-machine will change to state *Standby*.

[1] http://patterns.projects.cis.ksu.edu

The verification can be started directly from the IDE, in which case the state-machine is compiled into a NuSMV model, NuSMV is run, and the results are parsed back and lifted to the DSL level. Part b) of Figure 7 shows the result. At the top we have the list of executed checks and their results, at the bottom we illustrate the lifted counter example for the (failed) custom verification condition. By clicking on a state of the counter example, the IDE highlights the corresponding state in the program.

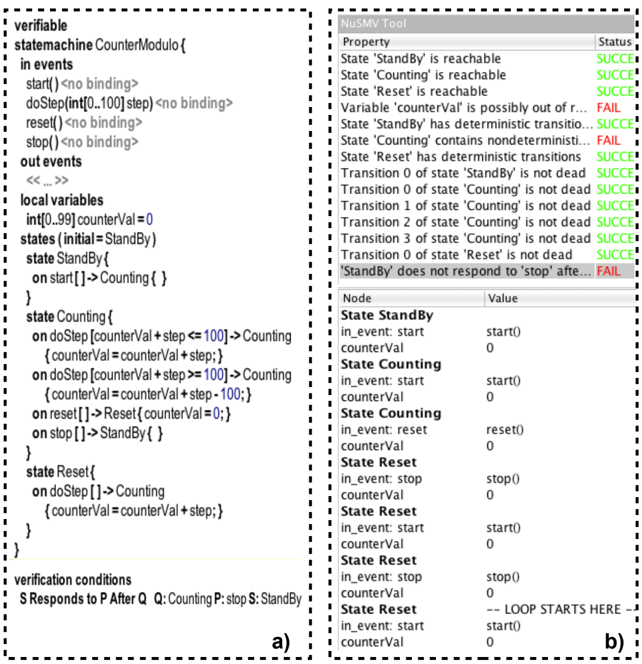

Figure 7. Model-checking the Statemachines:

Restricting the State Machines Language: By marking a statemachine as *verifiable*, a set of constraints and type-checking rules are performed that make sure that only the state machine language fragment is used that can be verified. Examples of constraints include: 1) we allow a maximum of one update of a local variable inside a transition; 2) support for only range and boolean types in local variables and input events and prohibiting integers, doubles or structured data types; 3) external functions may only be called via out-event bindings (the code inside these external functions is not part of the verification and we decouple it via the bindings).

The purpose of these restrictions is to enable straightforward transformation (of the state machine to NuSMV input) and result interpretation (of the NuSMV result at the DSL level). As with the decision tables, users of mbeddr can now choose between using a highly expressive variant of state-machines (and thereby losing the analyzability of their code) or using a restricted language that is verifiable.

C. *Consistency Checking of Feature Models*

Feature models are a well-known formalism for expressing product line variability at the domain level, i.e. independent of implementation artifacts (in the problem space as opposed to the solution space) [10]. A feature is essentially a configuration option. A feature model is a hierarchical collection of features, with constraints among them. Constraints include *mandatory* (a feature *must* be in each product configuration), *optional* (it *may* be in a product), *or* (*one or more* features of a set of features must be in a product) and *xor* (*exactly one* from a set of features must be in a product). In addition, there may be arbitrary cross-constraints between any two features (*requires-also* and *conflicts-with*). Feature models are often expressed via feature diagrams. In mbeddr we use a textual notation.

A *configuration* of a feature model is a valid selection of the features in a feature model. A *valid* configuration may not violate any of the constraints expressed in the referenced feature model.

Analyzing Feature Models: There are two obvious analysis in this context. The first one is checking feature model for consistentency, i.e. checking whether the set of constraints allows the definition of valid configurations at all. Conflicting constraints may prevent this (A *requires* B, B *conflicts with* A). The second analysis checks a specific configuration for compliance with its feature model. Both of these analyses are easy to perform with the help of SAT solvers [12] such as Yices.

Figure 8 shows an example of a feature model (a), of its translation to Yices (b), of the results of running Yices (c) and of the analysis results lifted to the domain level (d). Each of the assert *ids* from the unsat core given by Yices corresponds to a constraint from the feature model. Thereby we are able to present the mbeddr user directly with a list of those constraints that are violated.

IV. Methodology for Defining the Analyses

We propose an agile approach for combining domain specific languages and language extensions with formal analyses. Our methodology takes advantage of language engineering techniques provided by state-of-the-art language workbenches and has the following main characteristics:

1) Create language fragments that can be easily analyzed and that are well integrated with the rest of the code; make sure that analyses results can be lifted back at the domain level. Incrementally define and enlarge the language fragment that is supported by the analyses.

2) Make users aware of whether a program is currently analyzable or not. Give users the choice between writing code that is analyzable (by using a restricted subset of the language) or not analyzable (by using the unrestricted, but more expressive language).

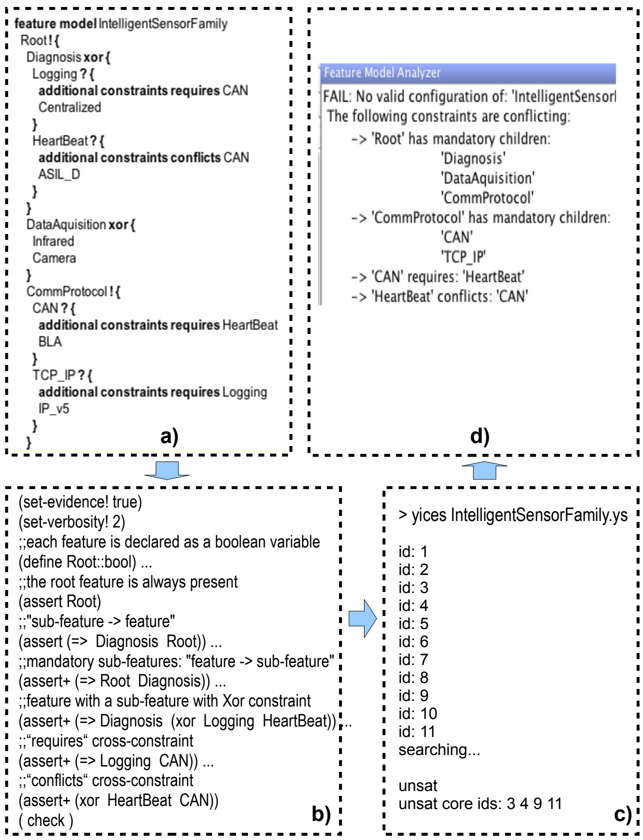

Figure 8. Analyzing Feature Models: a) feature models DSL; b) fragment of encoding feature models in Yices; c) results of running Yices; d) lifting analysis results at the domain level by mapping the unsat core to the intention behind the unsatisfied asserts

3) Design the analyses with the user in mind: enable them to easily write properties and lift the analysis results such that they are easy to interpret.

4) Decouple the analysis tool from the analysis use cases. An analysis tool can be used for a wide range of analyses of conceptually different C extensions.

Our proposed methodology has the following steps:

1) Choose the set of analyses that will be performed and at the same time decide on the language abstractions that are necessary to make the analysis feasible and run in a reasonable time. This may include the restriction of existing languages or the definition of additional abstractions.
2) Choose an analysis tool which will perform the actual analysis, translate the DSL fragment and the DSL-level properties in the modeling and specification language of the analysis tool
3) Lift the analysis results back at the domain level

V. Discussion

Soundness: Defining the analyses in an agile manner, based on domain-specific language extensions that inherently do not have formally defined semantics can easily lead to unsound analyses. It might not always be clear what exactly is verified: since from a DSL we translate to the analysis tool and to target language, keeping the two transformations consistent is challenging. We currently mitigate this issue by manual reviews and some automated tests.

Incrementality: Our approach is based on the definition of a subset of a language that, if users stick to it, enables advanced formal analyses. Our approach is incremental both in defining the analyses as well as in using them. At first, the supported language fragment can be kept small, and over time, if the need arises, the fragment can be enlarged by allowing more constructs. The trade-offs between the complexity of implementing the analyses, their usefulness to end users, and the size of the supported language fragment can be continuously evaluated. End users can decide whether to use a restricted language fragment that is analyzable or to use a more expressive fragment and thereby losing the analyzability. While the language extensions are modular, they are nonetheless integrated into the code-based IDE, enabling a smooth migration between the two choices, and no tool integration headaches.

VI. Related Work

The use of (formal) analysis techniques for GPLs is supported by a range of tools; most prominent are static analyzers for run-time errors like Polyspace [14] or Spec# [13]. However, experiences show that a substantial amount of annotation is often needed to capture constraints, which are lost when implementing higher-level concepts like state transition systems. Avoiding GPLs, on the other hand, altogether and resigning completely to DSLs like state transition systems, has proven to often be impractical as these language fragments often limit the expressibility, e.g., of actions, too much.

Therefore, in contrast to the work cited above, the presented approach demonstrates the advantages of using a modularization of the implementation language, combining the analyzability of restricted DSLs with the the expressiveness of GPLs. Leaving the choice to the language user, an individual trade-off is possible to ease practical application.

Formal Analyses: The analyses we have presented in this paper have been well known in the literature for many years. Our contribution is the integration of these analyses with language engineering technologies. Thereby we hope to contribute to a wider application of formal methods with practitioners. [4] is an early work that translates a fragment of the StateCharts language into SMV. [2] presents in detail meta-properties of state machines that represent vulnerabilities and defects introduced by developers that can be automatically verified. The properties are classified into minimality, completeness and consistency and are similar to the default properties that we check.

Our analysis method of feature models is similar to the one presented by Mendonca et. al. in [12].

[7] proposes an approach for defining and analysing tabular expressions in Simulink similar to our analysis of decision tables. In addition to using an SMT solver (as we do as well); they also use a theorem prover, mainly for dealing with non-linear expressions.

Correct-by-Construction: [8] defines a methodology for constructing programs that are analyzable. In the correct-by-construction methodology programs are continuously checked by using only polynomial time algorithms. In this manner the verification is done continuously and is integral part of the development process. The mbeddr technology stack can be seen as a pragmatic operationalization of the correct-by-construction approach where the analyzable language fragments are incrementally extended.

Usability of Formal Analyses: [11] characterizes the challenges in three categories: firstly, it is difficult to formalize the problem in the language of the verification tool (known as the model construction problem); secondly, it is difficult to formalize the properties to be verified, and, finally, once the result is obtained (at the abstraction level of the verification tool) it is difficult to lift it and interpret it at the domain level. All these challenges are due to the gap between domain specific abstractions and how they are reflected in programs on the one hand, and the abstractions of the analysis tool on the other hand. With deeply integrated analyses in mbeddr we tackle many of these challenges.

VII. Conclusions and Future Work

In this paper we propose a novel approach for pursuing formal analyses based on language engineering technologies. Specifically, we define language fragments that are "easily analyzable" and embed them in C. This way, when implementing a (part of a) program, developers can choose between either using a more restricted language and thereby *gaining* analyzability, or using a more expressive language and thereby *losing* analyzability.

This way we empower and encourage developers to write code in high-level and expressive DSLs that are appropriate for the problem at hand, while remaining in a fundamentally code-based environment. We further provide a set of DSL-specific out-of-the-box analyses that are based on the fundamental abstraction of the problem domain. The analysis results are lifted back to the domain level made available in the IDE, making the results much simpler to interpret.

Our future work is focused on two directions. First, at the framework level we plan to extend infrastructure for the definition of languages and analyses, with focus on the assurance of consistency between the translation of the DSL to the target language and to the analysis tool. Second, we plan to integrate new analysis tools that provide a high degree of automation, to add new analyses to the existing languages and to explore new languages and the relevant analyses.

References

[1] The yices homepage, http://yices.csl.sri.com.

[2] P. Arcaini, A. Gargantini, and E. Riccobene. Automatic review of abstract state machines by meta property verification. In *Proceedings of the Second NASA Formal Methods Symposium*, pages 4–13. NASA, 2010.

[3] A. Cimatti, E. M. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. Nusmv 2: An opensource tool for symbolic model checking. In *CAV*, 2002.

[4] E. M. Clarke and W. Heinle. Modular translation of statecharts to smv. Technical report, Carnegie Mellon University, 2000.

[5] J. C. Corbett, M. B. Dwyer, J. Hatcliff, S. Laubach, C. S. Păsăreanu, Robby, and H. Zheng. Bandera: extracting finite-state models from java source code. In *ICSE*, 2000.

[6] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett. Patterns in property specifications for finite-state verification. In *ICSE*, 1999.

[7] C. Eles and M. Lawford. A tabular expression toolbox for matlab/simulink. In *Proceedings of the Third international conference on NASA Formal methods*, 2011.

[8] E. K. Jackson and J. Sztipanovits. Correct-ed through construction: A model-based approach to embedded systems reality. In *ECBS*, pages 164–176, 2006.

[9] R. Janicki, D. L. Parnas, and J. Zucker. *Tabular representations in relational documents*, pages 184–196. Springer-Verlag New York, Inc., 1997.

[10] K. Kang, S. Cohen, J. Hess, W. Nowak, and S. Peterson. Feature-oriented domain analysis (foda) feasibility study. Technical report, SEI, 1990.

[11] K. Loer and M. Harrison. Towards usable and relevant model checking techniques for the analysis of dependable interactive systems. In *ASE*, 2002.

[12] M. Mendonca, A. Wąsowski, and K. Czarnecki. Sat-based analysis of feature models is easy. In *SPLC*, 2009.

[13] D. A. Naumann and M. Barnett. Towards Imperative Modules: Reasoning about Invariants and Sharing of Mutable State. In *19th IEEE Symposium on Logic in Computer Science*, pages 313–323. IEEE CS, 2004.

[14] K. Wissing. Static Analysis of Dynamic Properties - Automatic Program Verification to Prove the Absence of Dynamic Runtime Errors. In *INFORMATIK 2007: Informatik trifft Logistik*, volume 110 of *LNI*, pages 275–279. GI, 2007.