Automated Domain-Specific C Verification with mbeddr

Zaur Molotnikov
Fortiss Institute
Guerickestraße 25
Munich, Germany
molotnikov@fortiss.org

Markus Völter
independent/itemis
Oetztaler Straße 38
Stuttgart, Germany
voelter@acm.org

Daniel Ratiu
Fortiss Institute
Guerickestraße 25
Munich, Germany
ratiu@fortiss.org

ABSTRACT

When verifying C code, two major problems must be addressed. One is the specification of the verified systems properties, the other one is the construction of the verification environment. Neither C itself, nor existing C verification tools, offer the means to efficiently specify application domain-level properties and environments for verification. These two shortcomings hamper the usability of C verification, and limit its adoption in practice. In this paper we introduce an approach that addresses both problems and results in user-friendly and practically usable C verification. The novelty of the approach is the combination of domain-specific language engineering and C verification. We apply the approach in the domain of state-based software, using mbeddr and CBMC. We validate the implementation with an example from the Pacemaker Challenge, developing a functionally verified, lightweight, and deployable cardiac pulse generator. The approach itself is domain-independent.

1. INTRODUCTION

The C programming language is used in many different application domains. On the one hand, engineers benefit from its flexibility and the lightweight execution environment. On the other hand, the low abstraction level of C makes C code hard to verify against requirements specified in application domain terminology. With existing C verification tools, the verification has to be performed at the abstraction level of C itself, which is tedious and error-prone. An example of an application domain-level requirement that must be fulfilled by software is given below (it applies to the cardiac pacemaker used as an example throughout this paper):

The pacer is configured with two timeout parameters, VRP and LRI. LRI varies between 500 and 1.500 in increments of 20, and VRP is at most 20% of LRI. In a given period of time (the tick), the pacer either senses a signal representing a natural heartbeat, or it does not. Before the VRP timeout such signals are to be ignored, after VRP they must be registered. If within the LRI timeout no heartbeat was registered, an artificial pacing must be performed.

ASE '14 Västerås, Sweden

Requirements such as this one describe the properties the system should have. To make them verifiable by tools, they have to be expressed as formal verification conditions. A (not necessarily correct) implementation of this pacing logic is given in Listing 1. This code could be verified using C verification tools such as CBMC [10], SATABS [11] or CPAchecker [7]. Working at the abstraction level of C, they can be used to, for example, check assertions or error-label reachability. This makes it impractical to directly represent the application domain-level semantics implied by requirements such as those described above [13, 30].

```
int t = 0;
bool makePace(Event e) {
  switch(e) {
    case Tick:
      if (t < LRI) {
        ++t;
        return false;
      } else {
        t = 0;
        return true;
      }
    case Sense:
      if (t < VRP) {
        return false;
      } else {
        t = 0;
        return false;
      }
  }
}
```

Listing 1: Sample pacing logic implementation in C

Another problem of verification on the level of the implementation language is the lack of language abstractions for specifying environments [35, 14]. The environment is the code (or more generally, a system) with which the system under verification (SUV) interacts; in particular the environment may represent the relevant aspects of the real world in which the SUV is designed to operate. The environment must be modeled as well. This model encodes assumptions under which the verification is performed. Such environments are usually nondeterministic (e.g., the heart may or may not beat at any given time) and constrained (i.e., the behavior is somehow bounded, usually as a consequence of characteristics of the real world; for example, there are physical limits as to how fast a human heart can beat). A well-defined environment can reduce the number of spurious counterexamples, and speed up the verification as a consequence of state space reduction [14]. Finally, assuming the verification conditions and the environment have been specified, the question remains of how to integrate them with the SUV to perform the verification. In other words, verification schemas must be specified. A verification schema can serve as a reusable, automated verification method that can be used with multiple different SUVs, verification

Listing 2 shows the system from Listing 1, implemented *and verified* using domain-specific C extensions. These extensions are developed with mbeddr, an extensible version of C

introduced in Section 2.2. The SUV is expressed as a state machine, which adds additional structure and domain-level semantics to the C code, while still containing C expressions and statements to cover the details. mbeddr's state machines have states as well as internal variables, jointly representing the *total state* of the state machine. The environment `Heart` is explicitly specified and contains nondeterministic code that interacts with the SUV. The SUV parameters `lri` and `vrp` are nondeterministically assigned and constrained, as specified in the requirements.

An inductive proof schema is used to verify whether a given SUV satisfies a set of verification conditions, starting from the initial set of total states `I`. The verification condition is specified using an `after/before/exists` temporal pattern, which is much closer to the original prose requirement (to pace at `LRI`) than the low-level encoding in C.

```
state machine Pacer {
  in Tick;
  in Sense;
  in Config(lri, vrp);
  out MkPace;
  int c, LRI, VRP;
  state Wait {
    on Sense[c<VRP]->Wait
    on Tick[c>=LRI]->Pace
    on Tick[c<LRI]->Wait{++c;}
    on Sense[c>=VRP]->Wait {
      c=0; }
} }

environment Heart { nondet smTrigger(Pacer, Sense); }

assign lri: 500 <= lri <= 1500 && lri % 20 == 0;
assign vrp: 0 <= vrp <= 0.2 * vrp;

total state set I for Pacer: smInState(Wait)
  && c == 0 && LRI == lri && VRP == vrp;

inductive for Pacer on Tick
  from: I
  environment: Heart
  conditions: after smIsInState(Wait)
              before smIsInState(Pace)
              exists c == LRI;
```

```
  state Init {
    on Tick->Wait {c=0;}
    on Config->Init {
      LRI = lri;
      VRP = vrp;
    }
  }
  state Pace {
    entry {send MkPace;}
    on Tick->Wait {c=0;}
  }
```

Listing 2: Sample pacing logic in mbeddr

We call the approach to use domain-specific extensions of C, tailored to verification purposes, domain-specific C verification (DSCV). Translation to verifiable C is used to perform the verification itself. Figure 1 compares our approach to the state of the practice in C verification tools such as CBMC. Today, a practicing developer has to manually bridge the abstraction gap between the application domain and the implementation by manually encoding the system and the to-be-verified properties in C (with the exception of some property-specific tools, see Section 6). He also has to interpret the low-level verification results in the context of the application domain. DSCV, in contrast, provides language extensions for these tasks. The SUV is encoded as a domain-specific *model*[1], and the verification conditions as well as the environment are expressed relative to the abstractions in the model. However, since the model is implemented using C *extensions* (and not with a separate modeling language), the full expressiveness of C remains available to the developer when needed. The verification results, obtained from the C-level verification tool, are lifted to the abstractions relevant to application

[1]In this paper, the term *model* always refers to the SUV expressed with application domain abstractions.

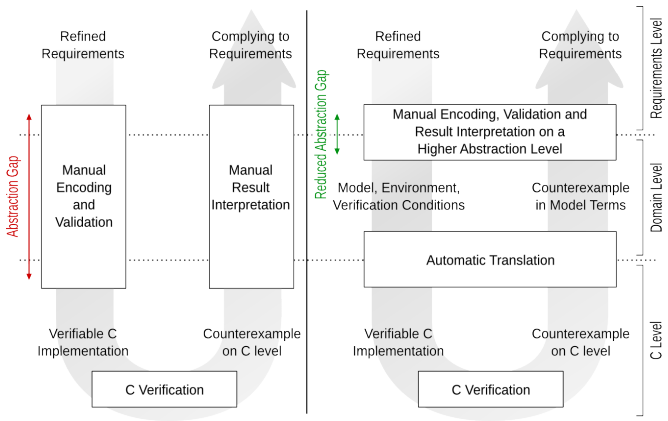

Figure 1: Todays C verification in practice (*left*) vs. DSCV

domain, closing the cycle. DSCV requires the following steps to obtain a verification tool tailored to specific application domain:

1. Create language extensions to model the system
2. Create extensions for verification conditions
3. Create extensions for the verification environment
4. Create a verification schema
5. Implement the interaction with the verifier

The use of language workbenches (discussed in Section 2.1) makes it straightforward to develop extensions for programming languages to support verification. Once built, the verification extensions can be reused to support verification of similar systems. New verification extensions can be built for fundamentally different systems in a modular way, i.e., without invasively modifying C or existing extensions.

Contribution This paper makes three contributions:

- The DSCV approach that simplifies verification of programs by using domain-specific language extensions. It combines existing C verification approaches with language engineering techniques
- An instantiation of DSCV for state-based software in C, relying on the CBMC verification tool and the extensible version of C provided by mbeddr
- A validation of DSCV based on the implementation and verification of a cardiac pacemaker pulse generator

From a user's perspective, the benefit of DSCV is that the expression, validation and maintenance of the SUV, the verification conditions and the environment is simplified due to closer alignment with the application domain. Verification schemas, support automated and reusable domain-specific verification methods. From the perspective of the verification developer, our approach enables modular, domain-specific verification in which new constructs for modeling SUVs, environments, verification conditions and for verification schemas can be added in a modular way. DSCV leverages the advances in C-level verification tools and brings them closer to the end-users through language engineering.

Previous Work This work builds on our previous research on using language engineering techniques for improving the usability of C-level verification [32, 31]. This paper advances

our research by introducing an end-to-end verification approach that combines high-level properties, advanced environments, and verification schemas to accomplish the complete verification of an executable and deployable subsystem.

Structure To implement DSCV we rely on a number of technologies, described in Section 2. Our implementation is based on the JetBrains MPS language workbench. It provides state-of-the-art language engineering facilities and is discussed in Section 2.1. To supply a version of C that can be easily extended towards verification we use mbeddr. A brief overview over mbeddr is provided in Section 2.2. We introduce CBMC and the connection between mbeddr and CBMC-based verification in Section 2.3. In Section 3 we describe how we have implemented DSCV for the domain of state-based software. This section is organized along the five steps required for implementing DSCV mentioned above. In Section 4 we discuss our DSCV-based contribution to the Pacemaker Challenge. Section 5 contains a discussion of the resulting verification tool and compares DSCV to other verification approaches. Section 6 discusses related work in the verification domain. We conclude the paper with a summary and a short outlook on our future work in Section 7.

Reproducing the Results Our work and results are reproducible because all components are open-source software. MPS can be downloaded from JetBrains.[2] The mbeddr source code can be obtained from the public Github repository.[3] The pacemaker example code is a part of mbeddr, it is in the `code/applications/Pacemaker` folder in the mbeddr sources. CBMC can be obtained from the CBMC website.[4] We also provide an installer[5] that ties all the components together, installing MPS, CBMC and building mbeddr from sources.

2. TECHNOLOGICAL BASELINE

2.1 Language Engineering with MPS

Our work relies on language engineering [37], which refers to defining, extending and composing programming languages and their integrated development environments (IDEs). Language workbenches [18] are tools that support efficient language engineering. Our implementation relies on the JetBrains MPS language workbench, which, unlike most other language workbenches [17], uses projectional editing.

Projectional Editing The conventional approach for language implementation relies on defining a grammar and then deriving a parser, which recognizes structure in the program text and materializes it into an abstract syntax tree (AST). An IDE is essentially a text editor which runs the parser incrementally to maintain the AST. The IDE also provides services such as syntax highlighting, navigation or code completion, and directly integrates the type checker and a compiler, if any. Many of these services rely on the AST. As discussed in [23], most grammar-based language engineering approaches and tools are limited in terms of modular extensibility or composability of grammars, because, depending on the grammar class, grammars are not closed under composition. While [23] also points out that purely declarative syntax definitions can address this challenge to a degree, there are still important cases where composability remains limited.

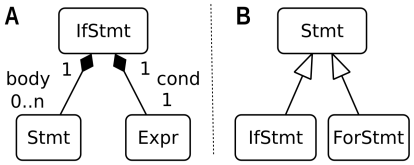

Figure 2: A fragment of language structure definition

Projectional editing is a different approach to defining, extending and composing languages and IDEs. A projectional editor does not rely on a parser. Instead, as a user edits a program, the AST is modified *directly*. Projection rules create a representation of the AST with which the user interacts, and which reflects the resulting changes (Figure 3 steps 1 and 2). As the user edits the program, program nodes are created as instances of language *concepts*. Concepts are the kinds of AST nodes, similar to metaclasses in traditional modeling approaches. In the editor, a code completion menu lets users create instances based on a text string entered in the editor called the alias. The valid aliases (and thus the concepts available for instantiation) are determined by the language definition. Importantly, *every next alias must be recognized as it is entered*, so there is never any parsing of a structured text sequence. In contrast to parser-based systems, where disambiguation is performed by the parser after a (potentially) complete program has been entered, in projectional editing, disambiguation happens at the time when the user picks a concept from the code completion menu: if two concepts define the same alias, the user resolves the ambiguity.

In a projectional editor every program node has a unique identifier and also points to its defining concept. So once a node is created, there is *never* any ambiguity what it represents, *irrespective of its syntax*. References between program elements are represented as references to the identifier. These references are also established during program editing by directly selecting reference targets from the code completion menu. This is in contrast to parser-based systems, where a reference is expressed as a string in the source text and a separate name resolution phase resolves the target AST element.

Defining and Extending a Language *Projectional editing makes it simple to define languages and IDEs, or extend them with new constructs.* Implementing a new language starts with the language structure, defining language concepts and the relationships between them. Figure 2A shows the example of an `IfStmt` that consists of a list of statements in its `body` and an `Expression` as the `condition`. Figure 2B shows that the `if` statement and the `for` statement concepts both inherit from `Stmt`. Similar to an object-oriented framework, MPS supports polymorphism, but it does so for language concepts. Consequently, `IfStmt` and `ForStmt` can be used in program locations where a `Stmt` is expected, for example, in the body of an `IfStmt`. In addition to concepts and their relationships, a language definition contains scoping rules, a type system and further structural constraints (not discussed here; see [36]).

Each language concept also has projection rules to define its representation in the editor. The notation can be textual, tabular, symbolic or graphical; mixing these notations is also possible, and a single concept can have several independently defined editors so that the notation of a program can be chosen by the user. In this paper we use mostly textual notations,

[2] `http://jetbrains.com/mps`
[3] `http://github.com/mbeddr/mbeddr.core`
[4] `http://www.cprover.org/cbmc/`
[5] `http://mbeddr.fortiss.org/download/`

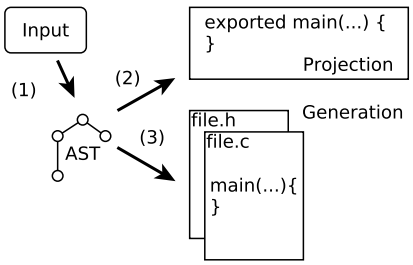

Figure 3: Projectional editing and code generation

because these are natural for C code. The textual notation of an AST in MPS should not be confused with real textual editing, however: it still uses the projectional approach. Examples of other notations used in mbeddr can be found in [38]. The particular details of defining projection rules are beyond the scope of this paper and are described in [37].

It is possible to extend languages without invasively changing their definition. An extending language defines new concepts and their relationship to existing concepts of the extended language, as in Figure 2. Consider extending C with an `unless` statement: we create a new language that contains an `UnlessStmt` concept that has a condition and a body as children, similar to the `IfStmt`, in Figure 2A. Next we make `UnlessStmt` inherit from the existing `Stmt` concept. We then define a projection rule that renders `UnlessStmt` as `unless (cond) {body}`. A typing rule ensures that the condition is Boolean. Since language definition in MPS always implies IDE definition, the `unless` statement will benefit from the same IDE services as the base language – the extensions integrate seamlessly. A systematic exploration of language extension and composition with MPS can be found in [36].

Generation and Transformation MPS languages are usually generated to real text at some point so they can be passed to existing compilers or verification tools. For this purpose, the definition of a base language such as C contains a text generator (Figure 3, step 3). For extensions, such as the `UnlessStmt`, no text generator is necessary because extensions are reduced to a semantically equivalent C representation before text generation is performed. Such transformations are defined between ASTs where nodes can be replaced with other nodes, additional nodes can be created and nodes can be removed. The `unless (cond) {body}` statement is reduced to `if (!cond) {body}`.

The set of transformations can form a cascade, which incrementally transform extensions into C, then into C header and implementation file modules, and finally into text files. The existing mbeddr generators deal with many details of textual C. For example, headers are automatically generated and name collisions are avoided by a name mangling mechanism. Since extensions are transformed to mbeddr C, extension developers do not have to care about these details. For details about defining transformations please refer to [37].

2.2 mbeddr

mbeddr is an open source project for embedded software development based on incremental, modular, domain-specific extension of C using language engineering technologies. Figure 4 shows an overview, details can be found in [38] and [39].

mbeddr and MPS mbeddr builds on MPS' language engineering facilities to define an ecosystem of over 60 languages.

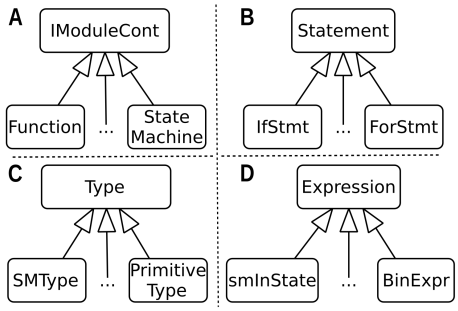

Figure 5: mbeddr concepts that act as extension points.

mbeddr specifically relies on language extension [36], where extending languages depend on an existing C base language and add additional constructs in a modular way.

mbeddr Languages As illustrated in Figure 4, mbeddr comes with an extensible implementation of C at its core. On top of that, mbeddr provides a library of reusable extensions useful for developing embedded software. As a user writes a program, he can import language extensions from the library. The main extensions include test cases, interfaces and components, state machines, decision tables and physical units for types and literals. For many of these extensions mbeddr provides an integration with verification tools (in particular, CBMC [10]). mbeddr also supports three important aspects of the software engineering process: requirements specification and tracing [40], product line variability, and documentation. We return to the requirements and tracing support when we briefly discuss validation of verified systems in Section 4.

mbeddr Extension Points To make typical language extensions simple to build, mbeddr comes with a set of abstract concepts and interfaces that act as extensions points: domain-specific extensions inherit from one of these extension point concepts (Figure 5). In the following we use state machines to illustrate some of these extension points because state machines play an important role in the pacemaker example.

`IModuleContent` represents module-level abstractions (Figure 5A). For example, `struct` declarations and functions, as well as mbeddr state machines inherit from `IModuleContent`. All "file-level" C extensions are module contents.

The imperative behavior of C is expressed as statement lists. If a new kind of statement is required, the `Statement` concept must be used as the parent (Figure 5B). For example, mbeddr provides the `smtrigger(event)` statement which triggers a state machine with an in event. Lists of statements can also be contained in other extensions, typically module contents or other statements. For example, state machines contain statement lists in entry actions of states. C expressions represent values.[6] New expressions can be created by extending `Expression`. For example, all binary operators in mbeddr C (`+`, `-`, or `*`) extend `Expression` transitively via `BinaryExpression` (Figure 5D). mbeddr supports a number of expressions related to state machines. One of them is the `smIsInState(state)` Boolean expression which tests whether a state machine is in the state passed as the argument. Expressions are used also in the verification conditions shown in Figure 7A and Figure 7C. Finally, every expression and variable in C has a type. Whenever the type system of C

[6]Even tough they are not pure because the evaluation of expressions can have side effects

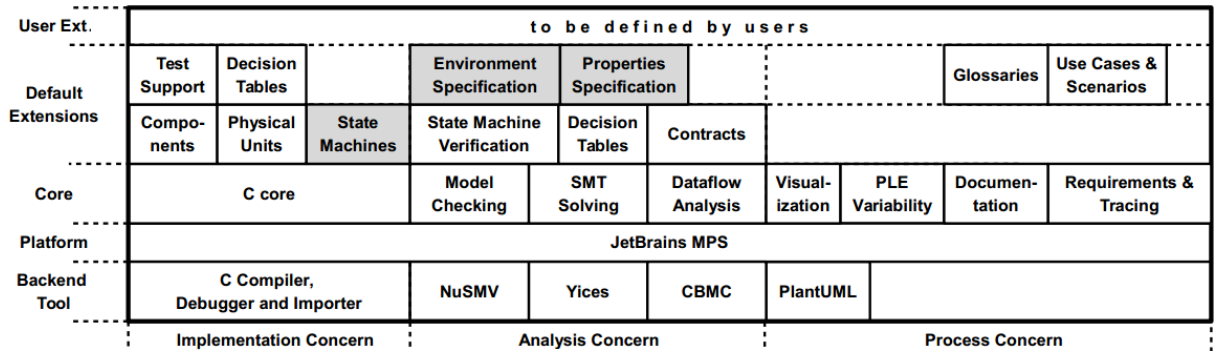

Figure 4: Layered architecture of mbeddr. It is based on MPS and provides a set of modular extensions to C (the ones most relevant for this paper are highlighted). At the back-end, it relies on established compilers and analysis tools.

has to be extended, a new type is defined that inherits from the `Type` concept (Figure 5C) and a typing rule is added.

2.3 C Verifiers and Language Engineering

C verifiers such as CBMC [10], SATABS [11] or CPAchecker [7] are powerful enough to perform verification of subsystems. However, they are not used in practice to their full potential due to, among other reasons, usability challenges.

C verifiers are designed to reason about properties of C code including built-in robustness checks, user-defined assertions as well as checking the reachability of error labels. The abstraction gap between application-level properties and C code makes it practically hard to verify application-level properties using C verifiers (cf. the *requirements specification problem* [12]). Encoding application-level properties, which are usually directly derived from application requirements and expressed with application domain terminology, as error labels or `assert` statements is tedious and error-prone [13, 30]. An additional obstacle in the context of C verification is the difficulty to define the boundary of a SUV, because C lacks effective means to define module boundaries.

More generally, an environment for verification must be specified (the problem is discussed for Java in [35]). Such environments are often nondeterministic. C verification tools offer ways to introduce nondeterminism typically via calls to non-defined functions, uninitialised variables or unset function parameters. `assume` statements can be used to constrain nondeterminism and specify assumptions on the environment. However, just like the specification of the verification conditions, these abstractions are code-oriented and do not relate to the application domain. Examples of application domain-related environment specification primitives include ranges for parameters, a nondeterministic choice construct, nondeterministic state machines or regular expressions to specify event sequences. **Language extension can be used to define new, application domain-specific language concepts for specifying verification conditions and environments.**

However, these still have to be used in a way that actually leads to a verification of the conditions, and hence proves the SUV correct relative to the environment (or not, illustrated with a counterexample). A specification is required which makes this process explicit. Problems like this one are typical for theorem proving: how to get from a set of explicit hypotheses to a proof of an implied conclusion? We have explained above that, using C verifiers, it is hard to express the hypotheses (an environment) and the conclusion (the given conditions hold). It is even harder to express a verification procedure and argue for its correctness, since C is not designed for such purposes. **Based on the application domain-level verification conditions and environments, additional language extensions can be used to express verification schemas.**

Finally, tool integration must be provided that simplifies understanding of counterexamples and enables fixing of defects detected by the verification. First, the verification tool has to be invoked and multiple invocations have to be orchestrated, combining their results. Second, the counterexample obtained from the C verification tool has to be presented at the level of the application domain, enabling users to trace the origin of a problem and perform the necessary fixes.

Summing up, we identify the following problems with state of the art C verification: specification of the verification conditions, the environment and the verification schema, as well as tool integration. As indicated above, these challenges can be resolved by defining application domain-specific C extensions targeted towards verification, and by providing a deep integration with C verification tools. To be able to formulate the verification conditions and the environment on the level of the application domain, it is also necessary to describe the SUV itself at that level, and corresponding C extensions have to be provided. These represent application domain semantics *directly*, as opposed to hiding them behind low-level C details. We do not see this as a disadvantage, though, because this makes the system implementation more maintainable, and the implementation can be validated more easily against the requirements.

3. DOMAIN-SPECIFIC C VERIFICATION

This section describes the implementation of DSCV on top of mbeddr for state-based software, illustrated with the pacemaker example. The description is organized along the steps introduced in Section 1. A detailed discussion of the pacemaker logic itself is beyond the scope of this paper, and we refer to the Pacemaker Challenge specification by Boston Scientific [8]. Pacemakers have different pacing modes; the VVI mode is used in this section because of its simplicity. We verify the more complex DDD pacing mode with mbeddr-based DSCV as well; see Section 4 for details.

3.1 Modeling the System under Verification

The first step when implementing DSCV is preparing language extensions to implement the SUV. This is necessary to align the abstraction level of the SUV with the application domain-level verification conditions and environment.

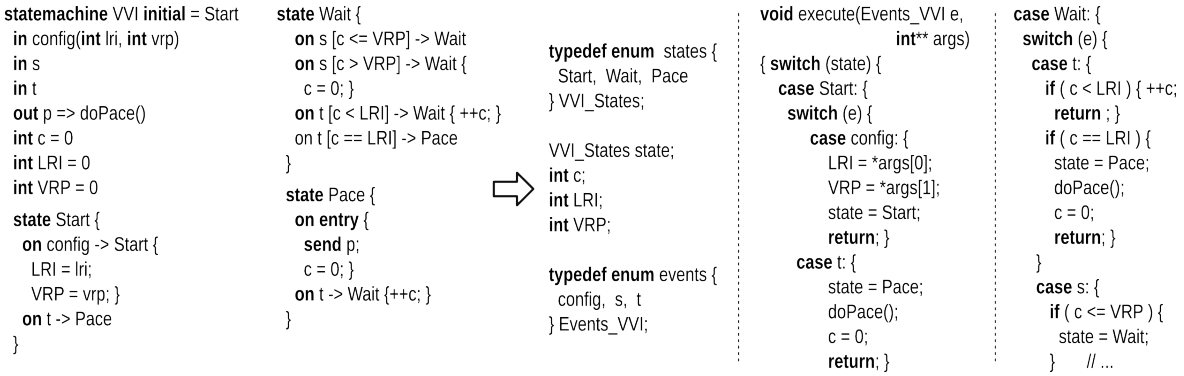

Figure 6: ***Left:*** **An example SUV model using a state machine.** ***Right:*** **The translation of the state machine to C.**

In many domains, requirements can be refined to finite-state automata; the pulse generator of a cardiac pacemaker is an example. We use mbeddr's existing state machine extension, so we do not have to define our own language extensions to model the SUV. In addition to enabling verification conditions at the abstraction level of state machines, the state machine also acts as the boundary of the SUV. An explicit system boundary is important for several reasons: it allows verification independent of the context in which the system is used; it makes reuse in a different context easier; and most importantly, the boundary makes explicit the SUV state space relevant for the verification processes. The boundary can be enforced via structural constraints and data flow checks, both available to the language engineer in MPS.

Figure 6 shows an example state machine and its (slightly simplified) translation to C that serves as an executable implementation and as a verification target for CBMC. States are translated to an `enum`, and variables represent internal data and the current state (the total state). A state transition function encodes the transitions. In the VVI pacing mode, the two parameters `LRI` and `VRP` specify when a pace should take place. They are set with the incoming `config` event (cf. the textual requirement at the beginning of the paper). An outgoing `p` event is bound to a function `doPace()`, which drives the pacer hardware. The reaction to incoming events depends on the current state, and the value of a counter `c` that is used in the transition guards; these are translated to `if` cascades.

If verification is performed in a domain for which mbeddr state machines fit well, they can be reused. Otherwise application domain-specific extension (incl. their transformation to C), have to be defined first. Section 2.1 explains how to do this and Section 2.2 introduces the typical extension points.

3.2 Verification Conditions

The second step in applying DSCV is providing language extensions to specify verification conditions. They should be aligned with the kinds of properties relevant for the SUV according to the requirements. Like the modeling extensions, the extensions for verification conditions should express application domain semantics, as opposed to C implementation details. The extensions have to be translated to constructs known to the C verifier, typically assertions or error labels.

If the model is a state machine, verification conditions usually relate to event sequences and states, as expressed by linear temporal logic (LTL) [6]. We have implemented direct support for a subset of the well-known specification patterns for finite-state verification introduced in [15]. These patterns are translated to monitor code blocks in C, verified by CBMC. Some examples of mbeddr's temporal patterns are:

- `after` *p* `until` *q* `must:` ***expr***
- `before` *p* `exists:` ***expr***
- `between` *p* `and` *q* `exists:` ***expr***

Figure 7A shows a pattern expressing the property that the counter `c` has to become the value `LRI` while in the `Wait` state, before the machine transitions into `Pace` state. In the translation to C (on the right side of the arrow), the flag `q` is used to start and stop setting the `e` flag, which in turn encodes the existential quantifier from the verification condition.

To support verification of state-based software, mbeddr provides abstractions to specify total state sets, nondeterministic initialization of state machines into a specified total state set as well as assertions on a machine state. We developed the history extension to store and query the event history of state machines specifically for the pacemaker. When verifying the more complex DDD pacing mode [8], these languages are all used together. Figure 7 shows examples.

The extensions for the verification conditions also rely on the extension points introduced in Section 2.2. For example, the temporal patterns inherit from `Statement`. They are translated into monitors and C verifier `assert` statements. New expressions are provided to query for properties of the modelling extensions. `IModuleContent` is used as the basis for the history language concepts and total state set definitions.

3.3 Verification Environments

Implementing and verifying software that works in an arbitrary environment is difficult and usually not necessary, because most software systems only have to work in a (more or less) well-defined environment. However, the assumptions and constraints implied in the environment must be made explicit. So the third step in implementing DSCV is the development of extensions for specifying environments.

Environments, by their nature, often exhibit nondeterminism. Unfortunately, C lacks language constructs to effectively define a nondeterministic yet constrained environment. And as with the system modelling and the verification conditions, the specification of the environment depends on the application domain: for a new domain mbeddr C must be extended with appropriate constructs.

For the pacemaker state machine, the following simple constructs are enough to create an overapproximation[7] of the real

[7]Overapproximated environments can exhibit more behaviors than the real environment. They can be easier to express, e.g., as an underspecifications of the real environments. Prov-

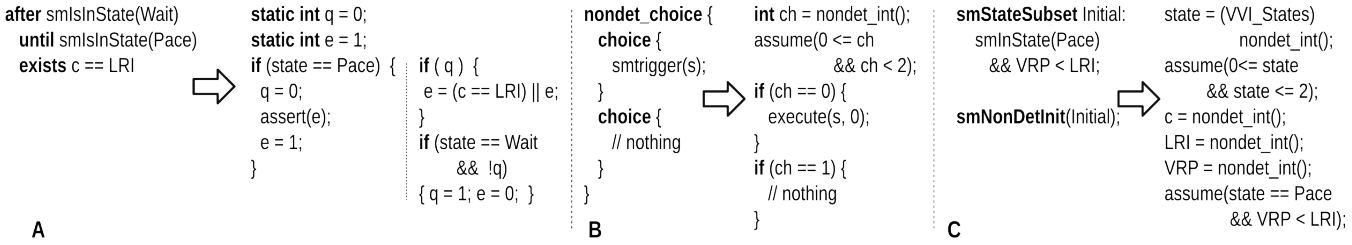

Figure 7: ***A:*** **A verification condition using a temporal pattern** ***B:*** **A verification environment using nondeterministic choices** ***C:*** **A verification schema element that uses nondeterministic initialization of a state machine.**

environment's behaviors. Figure 7B shows a nondeterministic choice construct, which either triggers an incoming event `s` on the state machine, or does nothing. This reflects the behavior of a human heart, which at any given time may produce a beat or not, nondeterministically. The `nondet_choice` construct is translated to a call to an undefined function, restricted by assumptions (illustrated below). A set of `if` statements execute behaviors based on the result of the choice.

A state machine with a nondeterministic choice in the transition guard could be used to specify a nondeterministic environment, which is itself state-based. The corresponding constructs are already available in mbeddr. Another useful mbeddr environment specification construct is a nondeterministic but constrained assignment of a variable, e.g., `nondet assign x constraint: x >= 10`. In the case of the pacemaker, the parameters are set (nondeterministically) by the medical doctor to a value that respects constraints defined by the system. Such an assignment is translated to a call to an undefined function and an `assume` statement. The code below shows the translation of the `nondet assign` given above:

```
x = nondet_int(); __CPROVER_assume (x >= 10);
```

Like other verifiers, CBMC provides function prototypes that can be called from user code to obtain nondeterministic data. These functions are implemented by CBMC itself.

3.4 Verification Schemas

The ability to model the SUV and to specify its environment and the properties does not automatically lead to a proof of the properties: the ingredients have to be used together in a consistent way to formulate a property proof obligation for the underlying verifier. Traditionally this is implied in the way the environment interacts with the SUV and the locations where the verification conditions are checked. DSCV uses language extensions to specify the proof obligation in an explicit and reusable way. We call this a *verification schema*.

The construction of the verification schema is also driven by the application domain. For a pacemaker, we assume a cyclic system where, in an interrupt service routine, a `tick` counter is incremented, an environment state is read, the events are triggered on the state machine accordingly, and then the conditions are checked. This process repeats forever. Figure 8 shows the verification schema for the pacemaker, as well as its translation to lower level verification primitives[8].

Bounded C verifiers such as CBMC can only verify a finite number of these read/trigger/check cycles. This means

ing a property P for the overapproximated environment leads to the conclusion that P holds also for the real environment.

[8]This is an example of a transformation cascade. The schema is not translated to C directly. Instead, it is translated to lower-level verification extensions which are themselves reduced to C by downstream transformations.

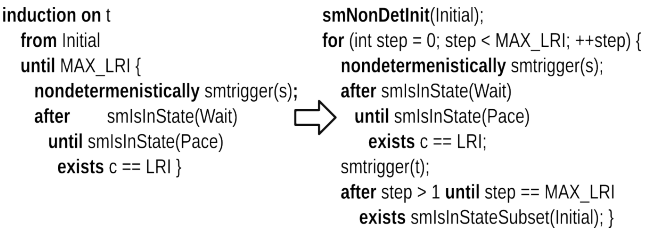

Figure 8: ***Left:*** **Inductive verification schema example and** ***Right:*** **Its translation**

that the conditions are verified only for a limited execution path of the system after its initialization. Additional effort is needed to verify the properties during the unbounded execution path. The verification schema for the pacemaker uses induction to achieve this:

1. An initial set of total states I is defined for the SUV
2. The SUV is nondeterministically initialized into I
3. In a loop, the environment interaction is performed and the properties are asserted
4. After a number of iterations, either the verification bound is met and the verification fails (because CBMC is a *bounded* model checker), or the SUV returns into I, meaning that the induction has succeeded.

If the CBMC bound was met, then one of the following steps can be taken: either the bound has to be increased, letting CBMC "try harder" to prove the induction. Alternatively, I must be redefined to be more limited, shrinking the state space the verifier must explore. Meeting the bound may also mean that the system really is *not* cyclic (i.e., a problem has been found). If the SUV returns to I, then, by induction, the asserted properties hold along any unbounded execution path of the SUV, given that it starts from a total state in I.

The last step in making the inductive proof valid is to show that the precondition (that the execution starts in I) is met. In the simplest case, the system is in I right after a regular, deterministic initialization. Otherwise it must be proven separately that the properties hold along the execution path that leads into I from the SUV's initial total state, into which the system is (deterministically) initialized during the actual execution. In such a case the verifier may have to be invoked more than once, and the tool integration (discussed in Section 3.5) must take care of orchestrating these invocations.

The inductive verification schema for the pacemaker (Figure 8) resembles an inductive proof on the state machine, as the translation on the right side of the figure shows. The induction is performed on the tick event `t`. At first a state machine is initialized nondeterministically[9] to a set of total states

[9]Figure 7C defines a total state set `Initial` and shows a

```
CBMC-based assertions analysis:
  entry point: DDD_Verification
  timeout: none (seconds)
  loops unwinding: 25
  unwinding assertions: false
  analysis depth: 10000
```

Figure 9: Analysis configuration in mbeddr

`Initial`, from which an inductive step starts, represented as a loop. The loop body consists of triggering nondeterministically the `s` event, which is equivalent to a `nondet_choice` from Figure 7B. Then the verification condition is asserted (Figure 7A), and finally `t` is triggered again. The `MAX_LRI` constant defines the maximum number of loop iterations until the induction step should converge, bringing the state machine back to the `Initial` set of total states. This is ensured with an assertion that checks that that the state machine is back in the `Initial` set of total states at some step.

As this example demonstrates, *verification schemas can be used to step beyond the capabilities of the underlying verification tool*, proving more or stronger properties. For example, the verification schema for the pacemaker described above overcomes the bounded nature of CBMC.

For the end user the schema implementation is hidden and verification is truly performed at the application domain-level. Once created, the verification schema can be reused with other SUVs, initial states and conditions. This reuse leads to lower effort for the verification of similar systems, making verification more feasible for the practitioner. In the case of the pacemaker verification, multiple pulse generator modes can be verified with the same verification schema.

3.5 Interfacing with the C Verifier

We have shown how application domain-specific languages such as state machines enable us to define higher level verification conditions, verification environments, and verification schemas. To perform the proofs we use the external C-level model checker CBMC. Two steps are necessary for integrating the verifier tool into mbeddr.[10] First, the verifier must be parametrized and invoked. Second, the C-level counterexamples produced by the verifier in the case where a property fails, must be lifted to the abstraction level of the domain.

Invoking the Verifier The problem of invocation and configuration is addressed with *analysis configurations*, another language extension (Figure 9). In addition to holding CBMC-related configuration parameters, they also identify the entry point, a function from which the verification starts. Since CBMC performs only relatively simple and isolated checks (e.g., based on label reachability), the tool integration also orchestrates the potentially multiple invocations necessary for verifying application domain-level properties.

To implement a new analysis configuration for CBMC, one can inherit from the `CBMCBasedAnalysisConfiguration` concept. A subclass of `Analyser` is created to orchestrate the calls to the verification tool. This is a rather mechanical task; however, a framework is provided to support extensibility.

Counterexample Lifting The actual verification happens at the level of C, and the counterexample obtained from the verifier is also expressed in terms of C. To make it meaningful to the user, it has to be lifted to the abstraction level of the SUV expressed as a model, its environment and verification conditions. Figure 10 illustrates how the counterexample is shown to the user: the view contains the failed property as well as the counterexample represented as a debug trace on the model level. Technically, the steps are bound to the lines of (generated) C code. As part of the lifting process, we shorten the CBMC output considering only the data that is relevant for showing the execution trace on the level of the application domain extensions: typically, several execution steps at the C-level can be lifted to a single execution step at the domain-specific language (DSL) level. The verification tool output is thus automatically compressed, becoming more accessible to the user (the highlighted assignment in Figure 10 corresponds to over 25 lines of XML verifier output). Relying on MPS, mbeddr keeps track of the mapping of the original program nodes to the final line numbers in the textual C representation. This enables us to bind the trace steps to the program nodes of the SUV. The UI supports navigation from the trace to the SUV code.

Property	Status	Trace ...	Time
Assert Checks	FAIL	338	18.16s

Idx	Raw	Kind	Value
33	0	var	5
34	0	ddd.mPVARP	5
35	76	**call**	generateEnvi...
36	0	res.AS_Detec...	1
37	0	res.VS_Detec...	0
38	0	env.AS_Detec...	1

Figure 10: Lifted counterexample in mbeddr

4. DSCV AND PACEMAKER CHALLENGE

Over the last years there have been many approaches to address the Pacemaker Challenge [8] by using high-level modeling languages and verification at the model level. However, none uses C directly to implement the pulse generator. The work in [29] demonstrates a tool chain that enables the translation of a verified UPPAAL model to Stateflow and then to C, applying it to a Pacemaker case study. In [25] the Vienna Development Method (VDM) is applied. In [33], the pacemaker is modeled using PROMELA and verified using the SPIN model checker; the corresponding C code, however, is hand-written based on a set of guidelines. In [20] Z is used to specify the pacemaker and to subsequently synthesize C code via the Perfect Developer tool.[11] Event-B is employed in [26] to formally model the pulse generator, EB2C generates C code from the Event-B model. In [24] the AADL language with BLESS annotations is used to model and verify a pacemaker VVI pacing mode. *In all cases listed above, the generated executable code, when obtained, is not verified.* The verification is performed only at the model level, and testing is used to check the C code. With DSCV, while supporting *application domain-level* models for the SUV, the verification conditions and the environment, *always verify the generated C code.*

Deployment We used DSCV and mbeddr to implement and verify one of the more complex pacing modes, called DDD, verifying all the functional properties it must fulfil. We then deployed the generated and verified code to an Arduino-based system, performing hardware in the loop tests. These tests revealed no faults in the implementation.

nondeterministic initialization of a state machine. `Initial` requires the state to be `Pace`, but keeps the values of `c`, `LRI`, and `VRP` unspecified, assuming that `VRP < LRI`.

[10] We consider here only non-interactive verifiers, which can run in a batch mode, CBMC is one of these.

[11] `http://www.eschertech.com/`

Parameter	Programmable Values	Incr.
Lower Rate Limit (LRI)	30-50 ppm 50-90 ppm 90-175 ppm	5 ppm 1 ppm 5 ppm
Ventricular Refractory Period (VRP)	150-500 ms	10 ms
...	...	...

Figure 11: Pacemaker parameters table with ranges, as taken from the requirements document [8].

The ability to deploy the verified code shows that the code is efficient enough to run on the intended target platform; this is critically important for real-world use. To the best of our knowledge, we are the only group in the Pacemaker Challenge that has implemented a fully functionally verified, executable, deployable, and lightweight DDD pacing logic.

Validating the System Verification, as discussed in this paper, refers to the automated checking of formally specified verification conditions relative to a system implementation and a well-defined environment. However, in the end, a system must be *validated*, not just verified. Validation ensures that the system reflects the original requirements, typically specified as prose. In the context of verification this means that the verification conditions must reflect the requirements. DSCV and mbeddr help with this aspect via application domain alignment and requirements tracing.

DSCV-based models, conditions and environment are easier to validate than the corresponding C code would be, because they are aligned more closely with the application domain. However, this can be taken further: the primitives for specifying environments can be combined to create more high-level environment constructs. For example, in the Pacemaker specification [8] many parameters are described in tables (Figure 11). Since mbeddr supports tabular notations, it is possible to create such tables in mbeddr and transform them into the primitives presented above. For instance, the first row of the table can be translated to:

```
nondet assign LRI constraint:
   (LRI >= 30 && LRI < 50 && LRI % 5 == 0)
     || (LRI >= 50 && LRI < 90)
     || (LRI >= 90 && LRI <= 175 && LRI % 5 == 0)
```

In addition, the physical units provided by mbeddr could be used to enhance the type system checks.

mbeddr supports a language for requirements specification that flexibly combines prose text and formal specifications [40]. For example, the tables mentioned in the previous paragraph can be embedded into a requirements document. Even in this case they can be translated into verification conditions. mbeddr also supports ubiquitous tracing. Any program node (expressed in any language) can be traced to requirements (specified using mbeddr or external tools such as DOORS[12]). This way any modeling, verification or environment artifact can be directly linked to the requirements, further aiding with validation.

5. DISCUSSION

On the trustworthiness of verification results. DSCV relies on extending C with specific constructs for verification, translating them to C and running a C verifier subsequently. The code that is generated from the SUV will subsequently be executed on the target platform (the verification conditions and the environment will not be deployed, of course). As a consequence, all C details are verified to the degree they are addressed by the verifier. At the same time, the specification is more concise and closer to the application domain, which makes it easier to validate (see previous section).

However, errors in the transformation of the extensions to C may lead to undiscovered problems and hence, bugs in the executing code. However, the separate translation of the model, the conditions and the environment increases the confidence in the verification result, since, for a systematic mistake, all three translations have to be consistently wrong. Providing verified transformations would be the ultimate solution. However, this is not feasible because neither C nor the extensions have formally defined semantics in mbeddr.

On hiding low-level details about the verification. The extension developer has to deal with the details of the low-level verification, since he has to develop the transformations and the lifting of the results. However, from the perspective of the application developer who uses existing extensions for a given application domain, the extensions hide the low-level details, with one exception: a developer still has to provide the correct parameters for executing CBMC. Failing to do so may lead to faulty verification, as a consequence of, for example, too short path lengths. So users still have to understand what CBMC does and how it has to be parametrized – but specifying and maintaining the model, the conditions and the environment is simplified. In addition, since the verification verifies C code, the scalability issues of C-level verification apply: depending on the structure and the size of the SUV, the verification may suffer from state explosion.

On the efforts for building the extensions. Once the domain-specific extensions are available for a given application domain, expressing models, conditions and environments becomes more efficient. However, the extensions have to be built first. Building the state machines extensions took the mbeddr team a few weeks. Development of the extensions for conditions and environments to verify state machines can be done in a few days. Both requires developers who have experience with developing languages with MPS; this in itself requires a few weeks to learn. Whether these few weeks and days are feasible in a given domain must be judged in each case separately. Like with all reusable assets, the decision depends on how often a set of extensions are expected to be used. Considering that the pacemaker contains many different pacing modes (implemented with different state machines and verification conditions) we think that the trade-off makes sense in this particular case.

On the end-user workflow. Assuming the extensions are available, we expect developers to use verification-driven development: the model, the conditions and the environment are evolved iteratively, in parallel. This is similar in spirit to test-driven development [4], and has two advantages. First, after each iteration, the SUV, the conditions and the environment are in sync, and the trust in the SUV builds over time. Similar to test-driven development, the repeated verification drives the design of the SUV to be modularized in a way that makes the verification feasible in terms of complexity and performance (smaller modules, well-defined interfaces). In contrast, experience tell us that it is often infeasible to verify a system *after* it has been implemented fully, *without* consideration for verification during the implementation process.

[12] http://ibm.com/software/products/en/ratidoor

On using DSCV with other domain-specific constructs. We have shown that state machines are a good basis for DSCV. However, the approach is not restricted to state machines. We are currently using DSCV for checking component contracts (discussed in [32]) and for checking the consistency of data flow models (as a part of a commercial tool for Siemens PL (LMS); we cannot disclose the details here).

On applying DSCV with other languages. We have integrated CBMC with mbeddr, leading to more domain-specific, and hence more usable verification. This is made possible by two ingredients. The first one is a verification tool (such as CBMC) that supports label reachability analysis and assertion checking in the target language. Reachability and assertions are useful because they support encoding of many other verification goals in the target language. The second ingredient is a way to efficiently define modular extensions (for models, conditions and environments) for the desired base language (C in our case). We conjecture that the same approach can be used for other languages if these two ingredients are available. For example, Java Pathfinder [21] can be used to perform reachability analysis for Java, and SugarJ [16] can be used to extend Java (and its IDE).

6. RELATED WORK

Code-level verification and specification: In comparison to existing ways of specifying verification conditions [3, 19, 9], DSCV enables their expression at the level of the application domain. Our work addresses directly the fourth challenge from [22]: *[..] to find effective ways to structure software such that formal verification techniques [..] become simpler to use and more effective in identifying potential violations of correctness properties in executable code.* mbeddr is extensible with new modeling constructs, verification conditions and environments for different domains or modeling formalisms; they can be used together in an integrated fashion, in the same development environment, and together with non-verified parts of a system. This makes DSCV and mbeddr more efficient to apply in practice because it eliminates the overhead in using and integrating different languages and tools. An additional advantage of DSCV is that there is no need to annotate the implementation code with additional verification-specific information (such as loop invariants in Frama-C [3]) because the semantics are directly implied in the extensions. This enables the clean separation of implementation and verification concerns and makes the verification more accessible. In [27] C program properties are encoded using a language that allows expressing LTL formulas with C boolean expressions over global variables. From these properties a monitor thread is generated and a model-checker is used for verification. An algorithm drives the interleaving between the monitor thread and the program to ensure that property checks are performed at all meaningful program locations. In contrast, DSCV requires the user to take care of interleaving monitors correctly with the code. Despite our example, DSCV is not restricted to LTL properties: users are encouraged to create their own high-level property specification extensions. Thus, the results of [27] could be exploited using DSCV by adding new languages for properties and generating monitor threads as in [27], and using a suitable model-checker. In [2], the low-level C-like SLIC specification language is used to express potential library misuse problems in C programs. SLIC makes it easy to encode simple temporal patterns. C code is generated from SLIC, and verification is used to detect property violations. Our approach differs by providing an extensible way to build and verify programs on the application domain abstraction level, rather than checking programs for a specific category of bugs.

High-level specification languages: When compared to tools for verifying models expressed in high-level specification languages such as Z [34] or Isabelle/HOL [28], DSCV benefits from close integration with C. First, the verified artifact (the generated C program) is a strict subset of the artifact that is deployed on the target device; some of the auxiliary code required for verification is not deployed. This reduces the specification and implementation effort (since it has to be done only once) and also reduces the potential for mismatches between specification and implementation. Second, in contrast to separate languages with downstream code generation, DSCV benefits from the close integration with C: low level code can be used where necessary and no semantic and tool integration problems arise when combining the to-be-verified code with the rest of the system.

Existing modelling and verification tools: In comparison to modelling and verification tools such as Scade [1], Simulink[13] or UPPAAL [5], DSCV is focused on C-level verification. It supports modular extensions of C for different application domains, and for each extension it provides suitable verification mechanisms including rich verification environments and schemas. In addition, since the conditions and environments are implemented as C extensions, it is possible to use C-code seamlessly together with the high-level extensions, as long as the SUV boundary remains intact (see Section 3.1).

7. CONCLUSION AND FUTURE WORK

The constantly improving C verification tools are not used at their full potential. In this paper we presented a novel approach that improves the practicality of these tools by using language engineering. We addressed the problems of specifying the verification conditions and verification environments in a way that is aligned with the application domain and provided verification schemas to specify proof obligations. We implemented our approach using mbeddr and CBMC in the domain of state-based software. We validated our DSCV implementation by building a functionally verified, lightweight, deployable cardiac pulse generator.

mbeddr's extensibility supports straightforward adaptation to other domains as well as other kinds of verification conditions, environments and schemas. DSCV can be used with other languages as long as modular language extension is possible and a verification tool is available for the language that supports assertions and label reachability analysis.

Our future work will proceed along the following lines: First, we will investigate the degree to which partial environment assumptions that are targeted towards the verification of individual properties can be specified. Second, we will investigate verification schemas and environments for composition of systems. Third, we will experiment with a verification cloud where remote verification agents execute the verifier continuously. Finally, we will use MPS' new support for graphical notations to represent state machines graphically, further simplifying validation.

[13] `http://www.mathworks.de/products/simulink`

8. REFERENCES

[1] P. A. Abdulla, J. Deneux, G. Stålmarck, H. Ågren, and O. Åkerlund. Designing Safe, Reliable Systems using Scade. In *Leveraging Applications of Formal Methods*, pages 115–129. Springer, 2006.

[2] T. Ball, B. Cook, V. Levin, and S. K. Rajamani. SLAM and Static Driver Verifier: Technology Transfer of Formal Methods inside Microsoft. In E. Boiten, J. Derrick, and G. Smith, editors, *Integrated Formal Methods*, volume 2999 of *Lecture Notes in Computer Science*, pages 1–20. Springer Berlin Heidelberg, 2004.

[3] P. Baudin, J. Filliatre, C. Marche, and et al. *ACSL: ANSI/ISO C Specification Language, http://frama-c.com/acsl.html*, 2012.

[4] K. Beck. *Test-driven development : by example*. Addison-Wesley, Boston, 2003.

[5] J. Bengtsson, K. G. Larsen, F. Larsson, P. Pettersson, and W. Y. 0001. UPPAAL - a Tool Suite for Automatic Verification of Real-Time Systems. In R. Alur, T. A. Henzinger, and E. D. Sontag, editors, *Hybrid Systems*, volume 1066 of *Lecture Notes in Computer Science*, pages 232–243. Springer, 1995.

[6] B. Bérard, M. Bidoit, A. Finkel, F. Laroussinie, A. Petit, L. Petrucci, and P. Schnoebelen. *Systems and software verification: model-checking techniques and tools*. Springer Publishing Company, Incorporated, 1st edition, 2010.

[7] D. Beyer and M. E. Keremoglu. CPAchecker: A tool for configurable software verification. In *Proceedings of the 23rd International Conference on Computer Aided Verification (CAV 2011, Snowbird, UT, July 14-20)*, LNCS 6806, pages 184–190. Springer-Verlag, Heidelberg, 2011.

[8] S. Q. R. L. Boston Scientific. PACEMAKER System Specification, http://sqrl.mcmaster.ca/pacemaker.htm, 2007.

[9] P. Chalin, J. R. Kiniry, G. T. Leavens, and E. Poll. Beyond Assertions: Advanced Specification and Verification with JML and ESC/Java2. In *Proceedings of the 4th International Conference on Formal Methods for Components and Objects*, FMCO'05, pages 342–363, Berlin, Heidelberg, 2006. Springer-Verlag.

[10] E. Clarke, D. Kroening, and F. Lerda. A tool for checking ANSI-C programs. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2004)*, volume 2988 of *Lecture Notes in Computer Science*, pages 168–176. Springer, 2004.

[11] E. Clarke, D. Kroening, N. Sharygina, and K. Yorav. SATABS: SAT-based predicate abstraction for ANSI-C. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2005)*, volume 3440 of *Lecture Notes in Computer Science*, pages 570–574. Springer Verlag, 2005.

[12] J. C. Corbett, M. B. Dwyer, J. Hatcliff, and Robby. A Language Framework for Expressing Checkable Properties of Dynamic Software. In K. Havelund, J. Penix, and W. Visser, editors, *7th International SPIN Workshop*, volume 1885 of *Lecture Notes in Computer Science*, pages 205–223. Springer, 2000.

[13] L. Cordeiro, B. Fischer, H. Chen, and J. Marques-Silva. Semiformal Verification of Embedded Software in Medical Devices Considering Stringent Hardware Constraints. In *International Conferences on Embedded Software and Systems*, ICESS, pages 396–403, 2009.

[14] P. Dhaussy, F. Boniol, J.-C. Roger, and L. Leroux. Improving Model Checking with Context Modelling. *Advances in Software Engineering*, 2012, Jan. 2012.

[15] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett. Patterns in property specifications for finite-state verification. In *Proceedings of the 21st International Conference on Software Engineering*, ICSE '99, pages 411–420, New York, NY, USA, 1999. ACM.

[16] S. Erdweg, T. Rendel, C. Kästner, and K. Ostermann. SugarJ: Library-based Syntactic Language Extensibility. In *Proceedings of the 26th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, volume 46 of *OOPSLA*, pages 391–406, New York, NY, USA, Oct. 2011. ACM.

[17] S. Erdweg, T. van der Storm, M. Völter, M. Boersma, R. Bosman, W. R. Cook, A. Gerritsen, A. Hulshout, S. Kelly, A. Loh, et al. The State of the Art in Language Workbenches - Conclusions from the Language Workbench Challenge. In *Software Language Engineering*, pages 197–217. Springer, 2013.

[18] M. Fowler. "Language Workbenches: The Killer-App for Domain Specific Languages?", 2005.

[19] A. A. E. Ghazi, M. Ulbrich, C. Gladisch, S. S. Tyszberowicz, and M. Taghdiri. JKelloy: A Proof Assistant for Relational Specifications of Java Programs. In *6th NASA Formal Methods Symposium (NFM)*, pages 173–187, 2014.

[20] A. O. Gomes and M. V. Oliveira. Formal Development of a Cardiac Pacemaker: From Specification to Code. In *Proceedings of the 2Nd World Congress on Formal Methods*, FM '09, pages 692–707, Berlin, Heidelberg, 2009. Springer-Verlag.

[21] K. Havelund and T. Pressburger. Model Checking JAVA Programs using JAVA PathFinder. *International Journal on Software Tools for Technology Transfer*, 2(4):366–381, 2000.

[22] G. J. Holzmann, R. Joshi, and A. Groce. New challenges in model checking. In *25 Years of Model Checking*, volume 5000 of *Lecture Notes in Computer Science*, pages 65–76. Springer, 2008.

[23] L. C. L. Kats, E. Visser, and G. Wachsmuth. Pure and declarative syntax definition: paradise lost and regained. In W. R. Cook, S. Clarke, and M. C. Rinard, editors, *Proceedings of the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2010*, pages 918–932, Reno/Tahoe, Nevada, 2010. ACM.

[24] B. Larson, P. Chalin, and J. Hatcliff. BLESS: Formal Specification and Verification of Behaviors for Embedded Systems with Software. In *5th International Symposium, NASA Formal Methods*, pages 276–290, 2013.

[25] H. D. Macedo, P. G. Larsen, and J. S. Fitzgerald. Incremental Development of a Distributed Real-Time Model of A Cardiac Pacing System using VDM. In *15th Intl. Symp. on Formal Methods, Aabo Akademi, Finland*, pages 181–197, May 2008.

[26] D. Méry and N. K. Singh. Formal Development and Automatic Code Generation : Cardiac Pacemaker. In *International Conference on Computers and Advanced Technology in Education*, ICCATE, Beijing, China, Dec. 2011.

[27] J. Morse, L. Cordeiro, D. Nicole, and B. Fischer.

Context-Bounded Model Checking of LTL Properties for ANSI-C Software. In G. Barthe, A. Pardo, and G. Schneider, editors, *In Intl. Conf. on Software Engineering and Formal Methods, (SEFM)*, volume 7041 of *Lecture Notes in Computer Science*, pages 302–317. Springer, 2011.

[28] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL: a proof assistant for higher-order logic*, volume 2283. Springer-Verlag, Berlin, Heidelberg, 2002.

[29] M. Pajic, Z. Jiang, I. Lee, O. Sokolsky, and R. Mangharam. From Verification to Implementation: A Model Translation Tool and a Pacemaker Case Study. In M. D. Natale, editor, *IEEE 18th Real-Time and Embedded Technology and Applications Symposium*, RTAS, pages 173–184. IEEE, 2012.

[30] H. Post, C. Sinz, F. Merz, T. Gorges, and T. Kropf. Linking Functional Requirements and Software Verification. In *17th IEEE International Requirements Engineering Conference*, pages 295–302. IEEE Computer Society, 2009.

[31] D. Ratiu, M. Voelter, Z. Molotnikov, and B. Schätz. Implementing modular domain specific languages and analyses. In *Proceedings of the 9th Workshop on Model-Driven Engineering, Verification and Validation*, MoDeVVa '12, pages 35–40. ACM, 2012.

[32] D. Ratiu, M. Völter, B. Kolb, and B. Schätz. Using Language Engineering to Lift Languages and Analyses at the Domain Level. In *5th International Symposium, NASA Formal Methods*, NFM, pages 465–471, 2013.

[33] A. Sharma. Towards a verified cardiac pacemaker. Technical report, National University of Singapore, School of Computing, Nov. 2010.

[34] J. M. Spivey and J. Abrial. *The Z notation*. Prentice Hall Hemel Hempstead, Hertfordshire, UK, 1992.

[35] O. Tkachuk, M. B. Dwyer, and C. S. Pasareanu. Automated environment generation for software model checking. In *18th IEEE International Conference on Automated Software Engineering*, ASE 2003, pages 116–129. IEEE Computer Society, 2003.

[36] M. Voelter. Language and IDE Development, Modularization and Composition with MPS. In *Generative and Transformational Techniques in Software Engineering IV, International Summer School, GTTSE 2011, Braga, Portugal*, LNCS, pages 383–430. Springer, 2011.

[37] M. Voelter, S. Benz, C. Dietrich, B. Engelmann, M. Helander, L. Kats, E. Visser, and G. Wachsmuth. *DSL Engineering – Designing, Implementing and Using Domain-Specific Languages*. CreateSpace Publishing Platform, 2013.

[38] M. Voelter, D. Ratiu, B. Kolb, and B. Schätz. mbeddr: Instantiating a Language Workbench in the Embedded Software Domain. *Journal of Automated Software Engineering*, 20(3):339–390, 2013.

[39] M. Voelter, D. Ratiu, B. Schätz, and B. Kolb. mbeddr: an extensible C-based programming language and IDE for embedded systems. In *Proc. of Conference on Systems, Programming, and Applications: Software for Humanity, SPLASH '12*, pages 121–140. ACM, 2012.

[40] M. Voelter, D. Ratiu, and F. Tomassetti. Requirements as First-Class Citizens. In *Proc. Modellbasierte Entwicklung eingebetteter Systeme IX, MBEES '13, Schloss Dagstuhl*, pages 44–49, 2013.