# System Verification through Program Verification*

Daniel Dietsch, Bernd Westphal, and Andreas Podelski

Albert-Ludwigs Universität Freiburg, Freiburg, Germany
{dietsch,westphal,podelski}@informatik.uni-freiburg.de

**Abstract.** We present an automatable approach to verify that a system satisfies its requirements by verification of the program that controls the system. The approach can be applied if the interaction of the program with the system hardware can be faithfully described by a table relating domain phenomena and program variables. We show the applicability of the approach with a case study based on a real-world system.

## 1 Introduction

When software for systems that interact with a physical environment is to be developed, the requirements are typically *system requirements*. That is, requirements stated in terms of *domain phenomena* observable in the physical environment. For instance, when a software shall realise the monitoring of a backup battery of a system, one requirement could be

> "For each point in time, the battery-low warning light is on if and only if the battery is low, i.e. the current battery voltage is below 6.6 V",

or, formalised using LTL (cf. Section 2)

$$\varphi_{batt} : \mathsf{G}(\textit{battery-low warning light is on} \iff \textit{battery is low } (V_{batt} < 6.6V)).$$

Here, "battery-low warning light is on" and "battery is low" are domain phenomena. Whether such a phenomenon is present or absent in a given point in time can be measured in the physical environment.

There is a need to assess whether a given program $P$ is correct, that is, whether the system $S$ executing $P$ satisfies the requirements $\varphi$. Even battery monitoring can be safety critical, for instance if it is the backup battery of a fire alarm system. An undetected battery failure can cause undetected fires in case of power outage, a false indication of battery failure causes unnecessary costs.

Tests can falsify correctness. If the program $P$ is executed on the system hardware and if measurements in the physical environment show a violation of a requirement $\varphi$, then $P$ is clearly not correct. But to establish that $P$ is correct without executing it on $S$ (for instance because $S$ is not yet built), we

* Partly funded by the Ministry of Science and Culture (MWK) Baden-Württemberg in project "Verbundprojekt Salomo" (www.salomo-projekt.de).

M. Butler and W. Schulte (Eds.): FM 2011, LNCS 6664, pp. 27–41, 2011.

face the problem that the program only operates on program variables, not on domain phenomena. On programs, we can only evaluate and analyse *software specifications*, i.e. properties of the evolution of program variables over time, but not system requirements. In this situation, we need a software specification $f(\varphi)$ such that $P$ satisfies $f(\varphi)$ if and only if $S$ executing $P$ satisfies $\varphi$.

Closely related to this problem is the work in [18], which provides a general framework to derive software specifications from system requirements. There, the authors explain how the transition from requirements to specifications can be done in a structured way by introducing additional domain assumptions ("breadcrumbs") that shift single requirements closer to the software. A sequence of breadcrumbs gives a so-called crumbtrail from requirements to specifications.

In the example, one helpful domain assumption would be that the battery-low warning light is attached to an output pin on the system hardware which is accessed via memory-mapped I/O by the variable `SCL` in the program and that the light is on if and only if $\texttt{SCL} = 1$.

[18] defines the correctness of breadcrumbs, but does not describe an automatable procedure to obtain them. In general, providing breadcrumbs is a highly creative act which involves insight into properties of the domain and the system design. For example, the breadcrumbs of the treatment control system or the two-way traffic light in [18] are clearly not obvious.

At the end of a crumbtrail, we find a software specification. In [18], the corresponding software is given in form of a high-level Alloy program which is then analyzed. In this paper, the software specification refers not to an Alloy program but to C code which will be executed by a system. This raises the additional challenge of dealing with real C code running on controller boards, such as the delay between reading inputs and providing outputs due to the computation phase.

In this work, we observe that there is a special class of systems and domain phenomena where breadcrumbs simply make explicit the relations between domain phenomena and program variables that can be assumed to be known by the programmer. For this class, we significantly ease the creation of the last breadcrumbs in the crumbtrail.

We identify premises under which we can conclude the satisfaction of the system requirements from results of model-checking the program against a software specification, which is obtained by transforming the system requirements. We argue that these premises are met at least by a certain common pattern of C programs and demonstrate the application on a case study employing a C verification tool.

Our approach allows for a high degree of automation. Except for providing the system requirements and the program, only the relation between domain phenomena and software observables is needed. The rest of the analysis can be carried out automatically.

We believe that especially small and medium-sized enterprises (SMEs) concerned with the development of safety-critical systems can benefit from this work. They typically cannot afford the high entry costs – in terms of training as well

as tool licenses – for the introduction of formal methods [8, 19]. Our approach reduces those costs while, at the same time, it allows for a gradual introduction of formal methods to development processes in SME.

The remainder of this document is organized as follows. Section 2 introduces the formal prerequisites of our approach, namely syntax and semantics of LTL with respect to Kripke structures. Section 3 details the formal foundation of the approach, Section 4 discusses its application to C programs of a certain form running on system hardware with memory mapped I/O or special function registers. In Section 5 we present the verification of an excerpt of a real world system, a radio-based fire alarm system, with the Verifying C Compiler [6, 21] as a case study. Section 6 discusses the related work and Section 7 summarises our contributions and names future work.

## 2 Preliminaries

### 2.1 Kripke Structure

A *Kripke structure* $M$ over a set of variables $Var$ is a tuple $M := (S, s_{init}, \rightarrow, \mu)$, where

- $S$ is a finite set of states,
- $s_{init} \in S$ is an initial state,
- $\rightarrow \subseteq S \times S$ is the transition relation, and
- $\mu : S \rightarrow (Var \rightarrow \mathcal{D}(Var))$ labels each state with a valuation of the variables, i.e. with a function which assigns each variable in $Var$ a value from the domain $\mathcal{D}(Var)$.

A *path* $\pi$ in $M$ is a sequence of states $s_0 s_1 s_2 \ldots$ such that $(s_i, s_{i+1}) \in \rightarrow$ for all $i \in \mathbb{N}_0$. We write $\pi(n)$ to denote the $n$-th state $s_n$ of $\pi$.

Furthermore, $\Pi_M(s)$ denotes the set of paths in $M$ with $s_0 = s$, i.e. all paths that start in $s$ and $\Pi(M) := \Pi_M(s_{init})$ denotes the set of paths of the Kripke structure $M$.

### 2.2 LTL Syntax

Let $Expr_{\mathbb{B}}(Var)$ be a set of boolean expressions over variables $Var$. The set of LTL formulas over $Expr_{\mathbb{B}}(Var)$ is inductively defined as follows.

$$\varphi ::= expr \mid \neg\varphi_1 \mid \varphi_1 \wedge \varphi_2 \mid \mathsf{X}\,\varphi_1 \mid \varphi_1 \,\mathsf{U}\, \varphi_2 \mid \varphi_1 \,\overleftarrow{\mathsf{U}}\, \varphi_2$$

where $expr \in Expr_{\mathbb{B}}(Var)$ and $\varphi_1$, $\varphi_2$ are LTL formulas.

### 2.3 LTL Semantics

Let $M$ be a Kripke structure over $Var$ and $\varphi$ an LTL formula over $Expr_{\mathbb{B}}(Var)$. Let $\mathcal{I}[\![expr]\!](\beta) \in \{\top, \bot\}$ be the interpretation of boolean expression $expr \in Expr_{\mathbb{B}}(Var)$ under valuation $\beta : Var \rightarrow \mathcal{D}(Var)$. We say that $M \models \varphi$ iff $\pi, 0 \models \varphi$ for all paths $\pi \in \Pi(M)$. The satisfaction relation $\pi, n \models \varphi$, $n \in \mathbb{N}_0$, is inductively defined as follows.

$\pi, n \models expr$ iff $\mathcal{I}[\![expr]\!](\mu(\pi(n))) = \top$.
$\pi, n \models \neg\varphi_1$ iff $\pi, n \not\models \varphi_1$.
$\pi, n \models \varphi_1 \wedge \varphi_2$ iff $\pi, n \models \varphi_1$ and $\pi, n \models \varphi_2$.
$\pi, n \models \mathsf{X}\,\varphi_1$ iff $\pi, n+1 \models \varphi_1$.
$\pi, n \models \varphi_1 \,\mathsf{U}\, \varphi_2$ iff there exists $j \geq n$ such that $\pi, j \models \varphi_2$ and $\pi, i \models \varphi_1$ for all $n \leq i < j$.
$\pi, n \models \varphi_1 \,\overleftarrow{\mathsf{U}}\, \varphi_2$ iff there exists $j \leq n$ such that $\pi, j \models \varphi_2$ and $\pi, i \models \varphi_1$ for all $j < i \leq n$.

## 3 The Interface between Requirements and Software

We consider programs to be Kripke structures over the program variables. We assume that there is a dedicated boolean program variable $v_{sn}$ which the programmer sets in the program whenever she considers the last inputs to be fully processed, where the outputs are stable, and where new inputs are read. For instance, the points in time where computed results are written into the memory-mapped I/O region or into special function registers to control output pins and where values of input pins are obtained via such addresses or registers (cf. Section 4).

**Definition 1 (Program).** *Let* $Var \supseteq \{v_{sn}\}$ *be a set of variables called* program variables*. The variable* $v_{sn}$ *is called* snapshot variable*.*

*A* program *over* $Var$ *is a Kripke structure* $P = (S_P, s_{init_P}, \rightarrow_P, \mu_P)$ *over* $Var \cup \{^\bullet v \mid v \in Var\}$ *where the snapshot variable is a boolean flag which holds in the initial state, i.e. for each* $s \in S_P$, $\mu_P(s, v_{sn}) \in \{\top, \bot\}$ *and* $\mu_P(s_{init}, v_{sn}) = \top$.

For simplicity, we assume that the valuation of $^\bullet v$ in a state $s$ provides the value of $v$ at the last snapshot state visited before $s$. We write $[expr]_{@\mathtt{pre}}$ to denote the expression obtained from $expr$ by syntactically substituting each variable $v$ by $^\bullet v$, i.e. the expression $expr[v := {}^\bullet v \mid v \in Var]$.

Given a set of program variables $Var$, an LTL formula over $Expr_{\mathbb{B}}(Var)$ is called *(software) specification*.

For us, a system $S$ is a hardware such as a controller board with inputs and outputs to which switches, sensors, etc. or lights, actuators, etc. can be connected, and a micro-processor which can execute programs. The named sensors and actuators interact with the environment which we consider not to be part of the system. For instance, a sensor can measure the voltage of a battery and an actuator can automatically dial a phone number to inform service personnel about power problems. Following Jackson et al. [18], the controller board, the sensors, and the voltage as well as the photons emitted by the light are part of the domain. A *domain phenomenon* is a phenomenon observable in the domain which is either present or absent. Such as the battery being low or the power warning light being on. Each domain phenomenon *dp* is either *controlled* by the system or *uncontrolled*. For example, the voltage of the battery is uncontrolled (an input to the system) while the warning light is controlled (an output of the system).

Given a set of domain phenomena $DP$, an LTL formula over $DP$ as atomic propositions (that is, by assuming that $DP$ is the set of variables, i.e. $Var := DP$, and that the set of expressions over these variables only provides the variable names itself, no logical connectives or functions, etc., i.e. $Expr_{\mathbb{B}}(Var) := Var$) is called *requirement*. Unless otherwise noted, from now on we assume that sets of domain phenomena $DP$ and boolean expressions $Expr_{\mathbb{B}}(Var)$ over program variables are disjoint.

Let $S(P)$ denote the behaviour of a system which is executing program $P$ in the considered domain, that is, $S(P)$ includes the evolution of domain phenomena over time. In general, $S(P)$ does not directly satisfy a requirement like the faithfulness of low battery warnings because the system in reality takes time to process the inputs. Such systems can in reality not process inputs in zero time. For example, there may be short periods in time where we can observe in $S(P)$ that the battery has recovered to a voltage above the critical threshold, that is, the domain phenomenon "battery low" is not observed, but that the warning light is still on because the program is currently processing the inputs. For the reasons given above, these violations cannot and should not be "blamed" on the program. So we consider a program $P$ to be correct if $S(P)$ already satisfies the requirements admitting a reasonable processing time to the program.

Instead of $S(P)$ we thus consider $S_\varepsilon(P)$ as representation of the system executing $P$, a Kripke structure which is (conceptually) obtained by observing $S(P)$ and noting down the presence or absence of controlled (output) domain phenomena and the presence or absence of uncontrolled (input) domain phenomena *at the last relevant point in time* according to some sampling procedure $\varepsilon$. Such a sampling procedure may depend on such various criteria as changes of the domain phenomena, the clock of the hardware board, or the current program counter. For example, if we use a predicate over the program counter as sampling procedure, we can separate the actual processing time of the program from the functional properties of the requirements. If necessary, techniques like worst case execution time analysis can be used to determine the actual time between the sampling points in the program and thus to conclude a time bound. Altogether, $\varepsilon$ ensures that in the states of $S_\varepsilon(P)$ we see the reaction of the system together with the last uncontrolled domain phenomena, on which the system reacted. Thereby, we can leave the formula from Section 1 unchanged and uncluttered by details of the (orthogonal) observation procedure. Over $S_\varepsilon(P)$, the example requirement $\varphi_{batt}$ correctly expresses that the warning light shall be on now if and only if the previous measurement of battery voltage found the battery to be low.

**Definition 2 (System).** *Let $DP$ be a set of boolean variables called* domain phenomena *and $P$ a program. A* system *controlled by program $P$ and observed according to sampling procedure $\varepsilon$ is a Kripke structure $S_\varepsilon(P)$ over $DP$.*

We say a program $P$ correctly realises the requirements $\varphi$ on hardware $S$ if and only if $S_\varepsilon(P) \models \varphi$.

**Definition 3 ($IRS$).** *Let $DP$ be a set of domain phenomena and $Expr_{\mathbb{B}}(Var)$ a set of boolean expressions over $Var$. A function*

$$IRS : DP \to Expr_{\mathbb{B}}(Var)$$

*is called* interface between requirements and software *(or $IRS$-table, for short).*

Each row of an $IRS$-table is a *domain assumption* in the sense of [18]. It states the assumption that domain phenomenon $dp \in DP$ is observable if and only if $IRS(dp)$ holds for a valuation of program variables. Intuitively, it makes explicit the programmer's assumption how, i.e. by which (expressions over) program variables, the program controls or obtains domain phenomena.

Note that $S_\varepsilon(P)$ is conceptually obtained by observing the running system so this structure is not directly available for analysis, in contrast to the program itself. Yet for the program, we cannot directly evaluate a requirement because a requirement is a formula over domain phenomena and a program only provides valuations of program variables.

Recall from the definition of programs that there is the dedicated snapshot variable $v_{sn}$ which indicates that the software has completely processed the last set of inputs. So we can apply the $IRS$-table backwards at each state of the program where $v_{sn}$ holds to obtain a system $\rhd_{IRS}$, i.e. a Kripke structure over domain phenomena. The program states where $v_{sn}$ does not hold are removed as they do not influence the environment and disregard the current environment situation. In the resulting $\rhd_{IRS}$, corresponding acceleration transitions are introduced between states $s$ and $s'$ if and only if there is a consecutive finite sequence of states $s = s_0 s_1 \dots s_n = s'$ in the program where $v_{sn}$ holds for $s$ and $s'$ but not in between.

**Definition 4 ($\rhd_{IRS}$).** *Let $P = (S_P, s_{init_P}, \rightarrow_P, \mu_P)$ be a program over $Var$, $DP$ a set of domain phenomena, and $IRS$ an $IRS$-table relating $DP$ and $Var$.*

*Then $\rhd_{IRS}(P)$ is the Kripke structure $(S, s_{init}, \rightarrow, \mu)$ over $DP$ with*

- $S = \{s \in S_P \mid \mu_P(s, v_{sn}) = \top\}$,
- $s_{init} = s_{init_P}$,
- $\rightarrow = \{(s, s') \in S \times S \mid \exists\, s_0, s_1, \dots, s_n \in S. s_0 = s \wedge s_n = s' \wedge$
  $\forall\, 0 \leq i < n. (s_i, s_{i+1}) \in \rightarrow_P \wedge \forall\, 0 < i < n. \mu_P(s_i, v_{sn}) = \bot\}$, *and*
- $\forall\, s \in S, dp \in DP. \mu(s, dp) = \mathcal{I}[\![IRS(dp)]\!](\mu_P(s))$.

We say program $P$ satisfies requirements $\varphi$ if and only if $\rhd_{IRS}(P) \models \varphi$.

Instead of applying Definition 4 in a constructive fashion, that is, constructing the Kripke structure $\rhd_{IRS}$ and checking requirement $\varphi$ on it, we want a software specification that is satisfied by a program if and only if the system executing the program satisfies the requirements. Given such a software specification, any program analysis procedure or tool able to decide whether the given formula holds for the program becomes directly applicable for deciding whether the program satisfies the requirements. In Definition 5 we give a procedure to construct such a software specification from a requirement, Lemma 1 states that the software specifications yielded by Definition 5 indeed characterises satisfaction of requirements.

**Definition 5** ($f^{v_{sn}}_{IRS}$)**.** *Let $P$ be a program over $Var$, $DP$ a set of domain phenomena, and $IRS$ an $IRS$-table relating $DP$ and $Var$.*

*Let $\varphi$ be a requirement, i.e. an LTL formula over $DP$. Then $f^{v_{sn}}_{IRS}(\varphi)$ denotes the software specification inductively defined as follows.*

$$f^{v_{sn}}_{IRS}(\varphi) := \begin{cases} \neg v_{sn} \mathbin{\overleftarrow{\mathsf{U}}} (v_{sn} \wedge IRS(dp)) & \text{iff } \varphi = dp \text{ (controlled)} \\ \neg v_{sn} \mathbin{\overleftarrow{\mathsf{U}}} (v_{sn} \wedge [IRS(dp)]_{@\mathtt{pre}}) & \text{iff } \varphi = dp \text{ (uncontrolled)} \\ \neg f^{v_{sn}}_{IRS}(\varphi_1) & \text{iff } \varphi = \neg\varphi_1 \\ f^{v_{sn}}_{IRS}(\varphi_1) \wedge f^{v_{sn}}_{IRS}(\varphi_2) & \text{iff } \varphi = \varphi_1 \wedge \varphi_2 \\ \mathsf{X}(\neg v_{sn} \mathsf{U} (v_{sn} \wedge f^{v_{sn}}_{IRS}(\varphi_1))) & \text{iff } \varphi = \mathsf{X}\,\varphi_1 \\ f^{v_{sn}}_{IRS}(\varphi_1) \mathsf{U} f^{v_{sn}}_{IRS}(\varphi_2) & \text{iff } \varphi = \varphi_1 \mathsf{U} \varphi_2 \\ f^{v_{sn}}_{IRS}(\varphi_1) \mathbin{\overleftarrow{\mathsf{U}}} f^{v_{sn}}_{IRS}(\varphi_2) & \text{iff } \varphi = \varphi_1 \mathbin{\overleftarrow{\mathsf{U}}} \varphi_2 \end{cases}$$

**Lemma 1.** *Let $P$ be a program over $Var$, $DP$ a set of domain phenomena, $\varphi$ a requirement, and $IRS$ an $IRS$-table relating $DP$ and $Var$. Then*

$$P \models f^{v_{sn}}_{IRS}(\varphi) \iff \rhd_{IRS}(P) \models \varphi.$$

*Proof.* By induction over the structure of $\varphi$ show the contraposition. □

By the following theorem, we can conclude from properties of the program $P$ to properties of the system $S_\varepsilon(P)$ if we employ a *valid* $IRS$-table, that is, an $IRS$-table which faithfully represents the dependencies between domain phenomena and program variables as observed when executing $P$ on a system $S$. In the particular case we consider here for simplicity, validity of $IRS$ implicitly requires that the observation procedure $\varepsilon$ corresponds to the processing of inputs as indicated by the snapshot variable.

**Definition 6.** *Let $P$ be a program over $Var$, $DP$ a set of domain phenomena, and $IRS$ an $IRS$-table relating $DP$ and $Var$. Let $S$ be a system and $\varepsilon$ an observation procedure.*

*The $IRS$-table $IRS$ is called* valid *if and only if $\rhd_{IRS}(P) = S_\varepsilon(P)$.*

**Theorem 1.** *Let $P$ be a program over $Var$, $DP$ a set of domain phenomena, $S$ be a system, and $\varepsilon$ an observation procedure. Let $IRS$ be a valid $IRS$-table relating $DP$ and $Var$.*

*Then for each requirement $\varphi$ over $DP$,*

$$S_\varepsilon(P) \models \varphi \iff P \models f^{v_{sn}}_{IRS}(\varphi).$$

*Proof.* Lemma 1. □

## 4 The Class of Memory-Mapped Systems

Memory-mapped systems are characterized by a close and direct interaction between input and output ports of the hardware and the memory the microcontroller provides to the program running on it. Typically there is a dedicated

area of memory (e.g. the first 512 bytes) where the program can read values that directly correspond to the applied voltage at an input port. In the same fashion there is a dedicated area of memory for the output ports of the hardware.

Furthermore, many memory-mapped systems operate in three phases. First, they read values from the input ports (read inputs), then they process the obtained data (process data) and finally they write the resulting values to the output ports (write outputs). After such a cycle, the system has reacted to changes in its environment and is stable again. Here we can observe if it adheres to its requirements or not and thus any sampling procedure $\varepsilon$ for such a system has to be interested in observing those stable states of the system. Naturally, an observation during the cycle, for example right after an input value has been read, would prohibit the system from exhibiting the correct reaction to the read input value and result in the system's failure to adhere to its requirements. More generally, systems can never react immediately (*atomic)* to changes in the environment because every system needs some time to calculate the reaction. Furthermore, the actual input values can change during the process-data phase or even during the write-outputs phase, again resulting in a mismatch between desired and observed system reaction.

Let us recall the battery measurement example from Section 1. There is a system with a backup battery. It has to routinely measure its battery to ensure it is operational in case of an emergency. The system reads the input providing the current battery voltage, calculates if this voltage is already too low and decides if it has to switch on the battery-low warning light and finally writes the actual reaction to the output variable. The sampling procedure $\varepsilon$ observes the state of the domain phenomena at the end of the cycle, compares the value read by the program (i.e. the memorized input value) to the defined battery-low threshold and observes if the light is on or not.

Now let us assume that our example system is controlled by a C program. Let us further assume that the hardware is such that the memory address `0xFF14` is mapped to the input port representing the battery voltage and the fourth bit of the word starting at `0xFF00` is mapped to the output port controlling the battery-low warning light. Then, we expect to see variable declarations similar to the ones shown in Figure 1.

Figure 2 shows the main function of the C program that together with the variable declarations from Figure 1 and the aforementioned hardware constitutes

```
1 //...
2 sfr P0 = 0xFF00;
3 //...
4 sfrp ADCR = 0xFF14;
5 //...
6 bool SCL = P0.3;
7 //...
```

**Fig. 1.** An example of the declaration of memory-mapped in- and outputs for an 8-bit microprocessor in a real-world C program. `sfr` and `sfrp` are compiler-specific keywords that indicate special function registers.

the system. The function is divided into two parts. First, all local variables are declared and initialized. Then the main loop follows, where the program progresses through the three phases, namely reading inputs, calculating a response to those inputs and finally writing outputs. After writing outputs, the program becomes stable and then the whole cycle starts again. Here the snapshot variable $v_{sn}$ evaluates to $\top$. Note here that the first evaluation of $v_{sn}$ to $\top$ is the initial state of the program $P$, thus we omit the initialization and the first execution of the loop body from our considerations.

Because the program completely controls the system, the three phases are exactly those we can observe when looking at the system. As we said, $\varepsilon$ samples the system at the end of every cycle. In the program, this is at the end of the `while`-loop, thus at the same point in time where $v_{sn}$ evaluates to $\top$. Therefore, we check for the desired behavior in the system as well as in the program at the same point in time, thus carrying the assumption $\rhd_{IRS}(P) = S_{\varepsilon}(P)$ from Definition 6 by ensuring that the sets of states of $\rhd_{IRS}(P)$ and $S_{\varepsilon}(P)$ have the same size. Although the identification of the three phases in general may be not as easy as in our example, a large class of programs, namely PLC programs (cf. IEC 61131 [9]), exhibits exactly those.

Another important aspect concerns the validity of the *IRS*-table. The variables in the program have to be bound to the "right" hardware addresses, that is, if the *IRS*-table states that the light is on if `SCL == 1`, `SCL` has to be bound to the output port controlling that light such that it is on if and only if `SCL` is set to 1. If this is the case can be easily validated by someone familiar with the hardware of the system. Consider a relation between program variables and hardware addresses, similar to the *IRS*-table (or the variable declaration shown in Figure 1). With such a relation and a description of the hardware, e.g. the data sheet of the microcontroller, one can formulate the relation between domain phenomena and hardware in terms of in- and output ports. This relation then states when a domain phenomenon is observable in terms of hardware in- and output ports.

To sum up, we say that if a system consists of a memory-mapping hardware and a program adhering to the three phases described above, and if the program variables used in the *IRS*-table are bound to the hardware addresses that correspond to the right domain phenomena, we fulfill the assumption $\rhd_{IRS}(P) = S_{\varepsilon}(P)$, which in turn enables the use of Theorem 1 to check the system requirement $\varphi$ directly on the program.

## 5 Case Study

Recall the requirement from Section 1:

$$\varphi_{batt} : \mathsf{G}(\textit{battery-low warning light is on} \iff \textit{battery is low } (< 6.6V))$$

We want to verify if the system described in Section 4 satisfies $\varphi_{batt}$. That system is actually an excerpt from the central unit F.BZ 100, which is embedded in the cc100 system, a radio-based fire alarm system which consists of F.BZ 100 itself,

different sensors and input/output devices as well as repeaters, all interconnected via high-frequency radio. In order to verify the system, we first need an $IRS$-table for the program shown in Figure 2. Figure 3 shows that $IRS$-table. Now we can apply the function $f_{IRS}^{v_{sn}}$ to $\varphi_{batt}$, which yields the following software specification:

$$\sigma_1 : \mathsf{G}\left((\neg sn \overleftarrow{\mathsf{U}} (sn \wedge \texttt{SCL == 1})) \iff (\neg sn \overleftarrow{\mathsf{U}} (sn \wedge \texttt{ADCR < 33152}))\right)$$

Since we already know that our system belongs to the class of memory-mapped systems, thus operates in three phases, we only need to show that our program adheres to the specification $\sigma_1$ to invoke Theorem 1 and be confident that the whole system satisfies $\varphi_{batt}$.

For our case we chose Microsoft's *Verifying C Compiler* (VCC) [6,21] as sound code-level analysis method. Because VCC has to handle the complexity of a low-level program with considerable size (Hyper-V [16] has approximately 100.000 LOC), we expected it to be usable for smaller system code as well. Besides a high level of automation and scalability, VCC also provides a tight integration in Microsoft Visual Studio [22], a commonly used integrated development environment (IDE). This integration allows an easy reporting of verification errors, comparable to the error messages provided by compilers [1].

The input to VCC is C code extended with annotations, which consist of function pre- and post-conditions, assertions, type invariants and specification code [6]. For VCC, we need to transform the software specification to annotations of the C program. This transformation depends in general on the form of

```
1  void main(void)
2  {
3    // init
4    bool led = 0;
5    unsigned int battery = 0;
6
7    // main loop
8    while(1)
9    {
10     // read inputs
11     battery = ADCR;
12
13     // calculate
14     led = (battery < 33152);
15
16     // write outputs
17     SCL = led;
18   }
19 }
```

**Fig. 2.** A program controlling a battery-low warning light. The variable `ADCR` is bound to the input port monitoring the battery voltage while the variable `SCL` is bound to the output port controlling the battery-low warning light.

| *DP* | $Expr_{\mathbb{B}}(Var)$ |
|---|---|
| battery low warning light is on | `SCL == 1` |
| battery low (<6.6V) | `ADCR < 33152` |

**Fig. 3.** The *IRS*-table for the requirement $\varphi_{batt}$

```
1  #include <vcc.h>
2
3  bool SCL;
4  volatile unsigned int ADCR;
5
6  void main(void)
7  writes(set_universe())
8  maintains(program_entry_point())
9  {
10   // init
11   bool led = 0;
12   unsigned int battery = 0;
13
14   spec(bool dp;)
15
16   // main loop
17   while(1)
18   {
19     // read inputs
20     battery = ADCR;
21     spec(dp = ADCR < 33152;)
22
23     // calculate
24     led = (battery < 33152);
25
26     // write outputs
27     SCL = led;
28
29     //check
30     assert(SCL <==> dp);
31   }
32 }
```

**Fig. 4.** The program from Figure 2 after preparing it for the code-level analysis by VCC. The highlighted parts are new annotations.

the software specification as VCC is not supporting LTL directly. For our experiments, we exploited the fact that the requirement is a simple global invariant.

A manual transformation of the specification yielded the program shown in Figure 4. In the following, we describe the important aspects of those annotations:

- In line 4 we declared the input variable `ADCR` as `volatile`. This causes VCC to interpret the variable as non-deterministic, i.e. it can assume every value its type allows regardless of the last write to the variable observed in the program, thereby representing all values the corresponding domain phenomena can assume. Also, note that because of this we could not use the variable declarations from Figure 1 but rather re-declared `SCL` and `ADCR`.
- In line 14 we declare a ghost variable `dp`, which is used in line 21 to memorize the value of the boolean expression $ADCR < 33152$ – representing the input domain phenomena "battery-low warning light is on" – directly after the read-access to `ADCR` occurred. This is done because, like in the real system, every access to `ADCR` could yield different values, but later in our check we want to use the value that actually determined the output. In this example `dp` corresponds to the ghost variables ${}^\bullet v$ from Definition 1.
- Finally, in line 30 we check if our specification holds. This is the only place where the last input is fully processed and therefore the snapshot variable $v_{sn}$ evaluates to $\top$. Here we place the `assert`-statement, thus synchronizing on the sampling procedure $\varepsilon$ of the system. We use that $v_{sn}$ evaluates to $\top$, so we can assert $SCL == 1 \iff ADCR < 33152$ to represent the software specification $\sigma_1$. We also substitute the memorized value for $ADCR < 33152$ from line 21 (`dp`) and get $SCL \iff dp$ as the actual condition that has to be checked.

Additionally we had to include the various VCC macros in line 1, declare that the function main may write and read every global variable (line 7) and that function main is the program entry point (line 8).

VCC reports that the `assert`-statement in line 30 holds, and with Theorem 1 and Section 4 we conclude that the system satisfies the requirement $\varphi_{batt}$.

## 6 Related Work

Constructive formal methods like RAISE [20], VDM [13] or the B-Method [15] have successfully been applied in industrial settings to construct correct systems (for a recent survey see [24]). They are centered around the stepwise-refinement paradigm [14], that is, the development starts by formulating high level requirements in a formal language and continues with the stepwise refinement of these. Every iteration adds more details to the formal representation and requires a new check for correctness. The process is repeated until the formal representation is detailed enough to allow a code generator to generate executable code. A tool that supports this activity is for example SCADE [7], which generates C or ADA code from the high-level language LUSTRE [5]. In contrast to our work, they employ a model-based development approach, which requires extensive knowledge for e.g. choosing the right abstractions. Such refinement-based approaches

typically require SMEs to restructure their whole software development process and to perform intensive training of the participating engineers.

Notably, the RAISE method provides a new approach to software engineering as a whole, namely domain engineering [4,3,2]. In domain engineering, one tries to formalize the important parts of the domain to define the bridge between domain and software automatically. Like our approach, it is concerned with the relation between domain phenomena and software representations, but instead of relying on the implicit domain knowledge of the developer manifesting in form of the $IRS$-table, it demands an explicit formalization of this knowledge. This, again, requires much effort and training from the user, but promises great benefits through the automatic detection of errors in the transition from the domain to the software. A recent example for the application of domain engineering can be found in [17].

The approach presented in this paper relates directly to previous work on the transformation of system requirements to software specifications, in particular [12] and [18]. In [12], conditions are given under which a transformation from requirements to specifications can be successful; the automation of the transformation is not considered. An extension of those considerations is presented in [18] in form of an iterative process called requirement progression. Its goal is to obtain a specification in terms of the to-be-specified software from requirements represented as problem frames [11]. There, domain assumptions – called breadcrumbs – are used to create a new requirement from an old one that talks about domain phenomena. In each iteration new breadcrumbs are introduced until the new requirement only talks about phenomena known to the software. In a sense, our $IRS$ function represents a special form of those breadcrumbs. We have for each domain phenomenon $dp$ an entry in the $IRS$, namely $dp \iff Expr_{\mathbb{B}}(Var)$. While this allows even untrained programmers to create such a function, it also prohibits the direct use in more complex systems, where a single domain phenomenon is not directly relatable to the software. Contrary to our approach, they do not consider C code which is still widely found in practice, thus their approach does not directly apply there.

## 7 Conclusion

We have presented an approach to verify that a program correctly realises system requirements by verification of the program code itself. The approach can be applied if the interaction of the program with the system hardware can faithfully be described by an $IRS$-table. The basic variant presented here already covers the huge class of systems where C code with dedicated read/process/write phases is executed on memory mapped I/O hardware and where domain phenomena are closely related to inputs and outputs of the system.

Given the system requirements, our approach requires nothing but the $IRS$-table and a C model-checker, in particular there is no need for a changed development process. We assume every programmer capable of developing software as discussed here is capable of creating an $IRS$-table because he or she is

necessarily already familiar with the domain phenomena in order to be able to develop the software.

Furthermore, we do not assume that system requirements are complete in any sense. Thus our approach can also be applied to only some, possibly most relevant system requirements giving control over the overall costs. By these advantages, our approach is in particular appealing for small or medium sized companies (SMEs), which on the one hand are often concerned with the class of systems we consider here but on the other hand cannot afford the high entry costs associated with formal methods or significant changes in the development process. Our approach provides for a gradual introduction of formal methods.

Furthermore, capturing the domain assumptions of the programmer in form of an $IRS$-table is a contribution towards dependability [10]. The $IRS$-table is an artifact which can be validated by domain experts. Additionally, the $IRS$-table defines a clear boundary of the responsibilities of the programmer. By providing the $IRS$-table, the programmer describes how sensors and actors have to be connected to, e.g., input and output pins of the system hardware. If the connection is according to the $IRS$-table, then the system will satisfy the system requirements. If the system malfunctions due to incorrect connections, this is clearly the responsibility of the party deploying the software.

The latter aspect is in particular relevant for sub-contracting software development. In addition to clearly limiting the responsibility of the programmer, it opens the possibility to decide the fulfillment of the contract by software model-checking tools [23].

Further work consists of a generalisation of the theory to more involved systems, e.g. where inputs and outputs have certain characteristics. For example, inputs may be constrained by environmental assumptions or can be fed back into the system directly or indirectly (such that there are dependencies between inputs and outputs). Also, outputs may not immediately change the environment but only after a certain delay.

Another important aspect is the automation and extension of C code annotations as described in Section 5, in particular covering the whole class of safety requirements in addition to the shown global invariant.

**Acknowledgments.** We would like to thank the anonymous reviewers for their many suggestions which helped us improving our work.

## References

1. Barnett, M., Chang, B.Y.E., DeLine, R., Jacobs, B., Leino, K.R.M.: Boogie: A Modular Reusable Verifier for Object-Oriented Programs. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2005. LNCS, vol. 4111, pp. 364–387. Springer, Heidelberg (2006)
2. Bjørner, D.: Domains as a prerequisite for requirements and software domain perspectives and facets, requirements aspects and software views. In: Broy, M., Rumpe, B. (eds.) RTSE 1997. LNCS, vol. 1526, pp. 1–41. Springer, Heidelberg (1998)

3. Bjørner, D.: Domain engineering: A software engineering discipline in need of research. In: Hlavác, V., Jeffery, K.G., Wiedermann, J. (eds.) SOFSEM 2000. LNCS, vol. 1963, pp. 1–17. Springer, Heidelberg (2000)
4. Bjørner, D.: Domain engineering: a "Radical innovation" for software and systems engineering? A biased account. In: Dershowitz, N. (ed.) Verification: Theory and Practice. LNCS, vol. 2772, pp. 100–144. Springer, Heidelberg (2004)
5. Caspi, P., Pilaud, D., Halbwachs, N., Plaice, J.: Lustre: A declarative language for programming synchronous systems. In: POPL, pp. 178–188 (1987)
6. Cohen, E., Dahlweid, M., Hillebrand, M.A., Leinenbach, D., Moskal, M., Santen, T., Schulte, W., Tobies, S.: VCC: A Practical System for Verifying Concurrent C. In: Berghofer, S., Nipkow, T., Urban, C., Wenzel, M. (eds.) TPHOLs 2009. LNCS, vol. 5674, pp. 23–42. Springer, Heidelberg (2009)
7. Halbwachs, N., Raymond, P., Ratel, C.: Generating Efficient Code From Data-Flow Programs. In: PLILP, vol. 22, pp. 207–218 (1991); special Issue on WOFACS 1998
8. Hall, A.: Realising the benefits of formal methods. J. UCS 13(5), 669–678 (2007)
9. IEC 61131 Programmable controllers, `www.iec.ch`
10. Jackson, D.: A Direct Path to Dependable Software. Commun. ACM 52(4), 78–88 (2009)
11. Jackson, M.: Software Requirements & Specifications: A Lexicon of Practice, Principles and Prejudices. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA (1995)
12. Jackson, M., Zave, P.: Deriving specifications from requirements: An example. In: ICSE, pp. 15–24 (1995)
13. Jones, C.B.: Systematic software development using VDM. Prentice Hall International (UK) Ltd., Hertfordshire (1986)
14. Kant, E., Barstow, D.R.: The refinement paradigm: The interaction of coding and efficiency knowledge in program synthesis. IEEE Trans. Software Eng. 7(5), 458–471 (1981)
15. Lano, K.: The B Language and Method: A Guide to Practical Formal Development. Springer, New York (1996)
16. Leinenbach, D., Santen, T.: Verifying the microsoft hyper-V hypervisor with VCC. In: Cavalcanti, A., Dams, D. (eds.) FM 2009. LNCS, vol. 5850, pp. 806–809. Springer, Heidelberg (2009)
17. Nami, M.R., Tehrani, M.S., Sharifi, M.: Applying domain engineering using raise into a particular banking domain. SIGSOFT Softw. Eng. Notes 32(2), 1–6 (2007)
18. Seater, R., Jackson, D., Gheyi, R.: Requirement Progression in Problem Frames: Deriving Specifications from Requirements. Requir. Eng. 12(2), 77–102 (2007)
19. Snook, C.F., Harrison, R.: Practitioners' views on the use of formal methods: an industrial survey by structured interview. Information & Software Technology 43(4), 275–283 (2001)
20. The RAISE Method Group: The RAISE Development Method. The BCS Practitioners Series, Prentice-Hall International, Englewood Cliffs (1995)
21. The Verifying C Compiler at Codeplex, `http://vcc.codeplex.com/`
22. Microsoft Visual Studio at MSDN, `http://msdn.microsoft.com/en-us/vstudio/default.aspx`
23. Westphal, B., Dietsch, D., Podelski, A., Pahlow, L.: Successful software subcontracting by system verification (submitted)
24. Woodcock, J., Larsen, P.G., Bicarregui, J., Fitzgerald, J.: Formal methods: Practice and experience. ACM Comput. Surv. 41(4), 1–36 (2009)