Abstract Interpretation-based Verification of Non-functional Requirements

Agostino Cortesi[1] and Francesco Logozzo[2] *

[1] Università Ca' Foscari di Venezia, I-30170 Venezia (Italy)
`cortesi@dsi.unive.it`
[2] École Polytechnique, F-91128 Palaiseau cedex (France)
`Francesco.Logozzo@Polytechnique.fr`

Abstract. The paper investigates a formal approach to the verification of non functional software requirements, e.g. portability, time and space efficiency, dependability/robustness. The key-idea is the notion of observable, i.e., an abstraction of the concrete semantics when focusing on a behavioral property of interest. By applying an abstract interpretation-based static analysis of the source program, and by a suitable choice of abstract domains, it is possible to design formal and effective tools for non-functional requirements validation.

1 Introduction

Abstract interpretation [10] is a theory of semantics approximation for computing conservative over-approximations of dynamic properties of programs. It has been successfully applied to infer run-time properties useful for debugging (e.g., type inference [7,28]), code optimization (e.g., compile-time garbage collection [22]), program transformation (e.g., partial evaluation [25], parallelization [36]), and program correctness proofs (e.g., safety [20], termination [5], cryptographic protocol analysis [33], proof of absence of run-time errors [3], semantic tattooing/watermarking [13]).

As pointed out in [30], there is still a large variety of tasks in the software engineering process that could greatly benefit from techniques akin to static program analysis, because of their firm theoretical foundations and mechanical nature.

In particular, as observed by [26], during the development of large-scale software systems, effective and efficient management of customer and user requirements is one of the most crucial, but unfortunately also least understood issues. Problems in the requirements are typically not recognized until late in the development process, where negative impacts are substantial and cost for correction has grown large. Even worse, problems in the requirements may go undetected through the development process, resulting in software systems not meeting customers and users expectations, especially when the coordination with other components is an issue. Therefore, methods and frameworks helping software

* This work was conceived when the author was visiting Ca' Foscari.

developers to better manage software requirements are of great interest for component based software.

In this paper, we are interested to investigate the impact of Abstract Interpretation theory in the formalization and automatic verification of Non-Functional Software Requirements, as they seem not adequately covered by most requirements engineering methods ([27], pag. 194). Non functional requirements can be defined as restrictions or constraints on the behavior of a system service [35]. Different classifications have been proposed in the literature [4,16,15], though their specification may give rise to troubles both in their elicitation and management, and in the validation process.

In fact, this work originated from a quite naive question: "*what do we mean when we say that a program is portable on a different architecture?*". In [17] a software is said portable if it can run in different environments. It is clear that it is assumed not only that it runs, but that it runs the same way. And it is also clear that if we require that the behavior is exactly the same, portability to different systems (e.g., from a PC to a PDA, or from an OS to another) can almost never be reached. This means that implicit assumptions are obviously made about the properties to be preserved, and about the ones that might be simply disregarded. In other words, portability needs to be parameterized on some specific properties of interest, i.e. it assumes a suitable abstraction of the software behavior. The same holds also for other product non-functional requirements, like space and time efficiency, dependability, robustness, usability, etc. It is clear that, in this context, the main features of abstract interpretation theory, namely modularity, modulability, and effectiveness may then become very valuable.

The main contributions of the paper can be summarized as follows:

– We extend the usual abstract interpretation notions to the deal with systems, i.e. programs + architectures.
– We show that a significant set of product qualities (non functional requirements) can be formally expressed in terms of abstraction of the concrete semantics when focusing on a behavioral property of interest. This yields an unifying view of product non-functional requirements.
– We show how existing tools for automatic verification can be re-used in this setting to support requirements validation; their practicality directly depends on the complexity of the abstract domains.

The advantage of this approach with respect to previous attempts of modelling software requirements, e.g. by using Milner's Calculus of Communicating Systems [19] or formal methods like Z [24] or B [1,2] is twofold: (i) the soundness of the approach is guaranteed by the general abstract interpretation theory, and (ii) the automatic validation process can be easily tuned according to the desired granularity of the abstraction.

As far as we know, this is the first attempt to apply Abstract Interpretation theory to the treatment of non-functional software requirements. These seminal results can be seen as a partial contribution towards the achievement of a more challenging objective: to integrate formal analysis by abstract interpretation in the full software development process, from the initial specifications to the ultimate program development [9].

Paper Structure In Section 2, the concrete semantics of a simple imperative language is introduced to instantiate our framework. In Section 3, the core abstract interpretation theory is extended to deal with program and architecture abstractions. In Section 4 we show how to instantiate our framework on a suite of non-functional product requirements. Section 5 concludes the paper.

2 Operational Semantics of a Core Imperative Language with Exceptions

In order to illustrate the results of this paper, we instantiate our framework with a core imperative language with exceptions and a core architecture. The results can be easily generalized to more complex languages and architectures. We give the syntax, the transition relations and the trace semantics of systems, composed by architectures and a programs.

2.1 Syntax

In this paper setting an architecture is a tuple $\langle \mathit{bits}, \mathit{Op}, \mathit{stdio}, \mathit{stdout}\rangle$, where *bits* is the number of bits used to store integer numbers, *Op* is a set of functions implementing basic arithmetic operations, *stdio* is the input stream (e.g., the keyboard) and *stdout* is the output stream (e.g., the screen). The input stream has a method *next* that returns immediately the next value in the stream, and the output stream has a method *add* to put a pair $\langle v, c\rangle$, i.e., a value v with a color c. We assume that if an arithmetic error occurs in the application of an operation $op \in Op$ (e.g., an overflow or a division by zero), then the exception `ExcMath` is raised.

The syntax of programs is specified by the following grammar:

$$\begin{aligned} \mathtt{C} ::= {} & \mathtt{skip} \mid \mathtt{x} = \mathtt{E} \mid \mathtt{C}_1; \mathtt{C}_2 \mid \mathtt{if(E!=0)}\, \mathtt{C}_1 \;\mathtt{else}\; \mathtt{C}_2 \mid \mathtt{while\ (E!=0)}\; \mathtt{C} \\ & \mathtt{write(x, col)} \mid \mathtt{throw\ Exc} \mid \mathtt{try}\; \mathtt{C}_1 \;\mathtt{catch(Exc)}\; \mathtt{C}_2 \\ \mathtt{E} ::= {} & k \mid \mathtt{read} \mid \mathtt{E}_1 + \mathtt{E}_2 \mid \mathtt{E}_1 - \mathtt{E}_2 \mid \mathtt{E}_1 * \mathtt{E}_2 \mid \mathtt{E}_1 / \mathtt{E}_2 \end{aligned}$$

where `x` and `col` belong to a given set `Var` of variables, `Exc` belongs to a given set `Exceptions` of exceptions (including the arithmetic ones) and k is (the *syntactic* representation of) an integer number.

A system is a pair $\langle \mathtt{A}, \mathtt{C}\rangle$, where `A` is an architecture and `C` is a program.

2.2 Semantics

The semantics of a system is described in operational style. We assume that the only available type is that of architecture-representable natural numbers:

$$\frac{\underline{k}\in\mathbb{N}_{bits}}{\langle k,\sigma\rangle\xrightarrow{\mathtt{E}}k}\qquad\frac{\underline{k}\notin\mathbb{N}_{bits}}{\langle k,\sigma\rangle\xrightarrow{\mathtt{E}}\langle\mathtt{ExcMath},\sigma\rangle}\qquad\frac{\mathtt{A}.stdio.next=v}{\langle\mathtt{read},\sigma\rangle\xrightarrow{\mathtt{E}}\langle v,\sigma\rangle}$$

$$\frac{\langle\mathtt{E}_1,\sigma\rangle\xrightarrow{\mathtt{E}}\langle v_1,\sigma\rangle\ \ \langle\mathtt{E}_2,\sigma\rangle\xrightarrow{\mathtt{E}}\langle v_2,\sigma\rangle\ \ v_1,v_2\neq\mathtt{ExcMath}\ \ \mathtt{A}.op(v_1,v_2)=v\neq\mathtt{ExcMath}}{\langle\mathtt{E}_1\,op\mathtt{E}_2,\sigma\rangle\xrightarrow{\mathtt{E}}\langle v,\sigma\rangle}$$

$$\frac{\langle\mathtt{E}_1,\sigma\rangle\xrightarrow{\mathtt{E}}\langle v_1,\sigma\rangle\ \ \langle\mathtt{E}_2,\sigma\rangle\xrightarrow{\mathtt{E}}\langle v_2,\sigma\rangle\ \ v_1,v_2\neq\mathtt{ExcMath}\ \ \mathtt{A}.op(v_1,v_2)=\mathtt{ExcMath}}{\langle\mathtt{E}_1\,op\mathtt{E}_2,\sigma\rangle\xrightarrow{\mathtt{E}}\langle\mathtt{ExcMath},\sigma\rangle}$$

$$\frac{\langle\mathtt{E}_1,\sigma\rangle\xrightarrow{\mathtt{E}}\langle v_1,\sigma\rangle\ \ \langle\mathtt{E}_2,\sigma\rangle\xrightarrow{\mathtt{E}}\langle v_2,\sigma\rangle\ \ (v_1=\mathtt{ExcMath})\ or\ (v_2=\mathtt{ExcMath})}{\langle\mathtt{E}_1\,op\ \mathtt{E}_2,\sigma\rangle\xrightarrow{\mathtt{E}}\langle\mathtt{ExcMath},\sigma\rangle}$$

Fig. 1. The transition relation for expressions

$\mathbb{N}_{bits}=\{0,,\ldots 2^{bits}-1\}$. Given the *syntactic* representation k of a number, $\underline{k}$ is the *semantic* correspondent. For instance, $\underline{\mathtt{0xFFFF}}=65535$ so that $\underline{\mathtt{0xFFFF}}\notin\mathbb{N}_{8}$. An environment is a partial map from variables to representable integers: $\mathsf{Env}=[\mathtt{Var}\rightarrow\mathbb{N}_{bits}]$. If a variable $\mathtt{x}$ is not defined in a state σ, we denote that by $\sigma(\mathtt{x})=\Omega$. A state is either a command to execute in a given environment, or an environment, or an exception raised within an environment. Formally: $\Sigma=\mathtt{C}\times\mathsf{Env}\cup\mathsf{Env}\cup\mathtt{Exceptions}\times\mathsf{Env}$.

The transition relations for expressions and programs are defined by structural induction, and they are depicted in Fig. 1 and Fig. 2. It is worth noting that the transition rules are parameterized by the underlying architecture (e.g., the raising of an overflow exception depends on $\mathbb{N}_{bits}$).

Let Σ^* denote the set of finite traces on Σ, and let $S_0\subseteq\Sigma$ be a set of initial states. With a slight abuse of notation, we refer to a state as a trace of unitary length. The partial-traces semantics [12] of a system is then expressed as a least fixpoint over the complete boolean lattice $\langle\mathcal{P}(\Sigma^*),\subseteq\rangle$ as follows:

$$\mathbb{S}[\![\langle\mathtt{A},\mathtt{C}\rangle]\!](S_0)=\mathrm{lfp}_{\emptyset}^{\subseteq}\lambda X.\ S_0\cup\{\sigma_0\ldots\sigma_n\sigma_{n+1}\mid\sigma_0\ldots\sigma_n\in X,\ \sigma_n\rightarrow\sigma_{n+1}\}.$$

3 Abstracting Systems = Programs + Architectures

Abstract interpretation [10] is a general theory of approximation which formalizes the idea that the semantics of a program can be more or less precise depending on the considered observation level. In this section we revise some basic concepts, and we extend them to deal with composed systems.

In the abstract interpretation terminology, $\langle\mathcal{P}(\Sigma^*),\subseteq\rangle$ is the *concrete domain*, its elements are semantic properties, and the order $\subseteq$ stands for the logical implication. As a consequence, the most precise property about the behavior of a system is the semantics $\mathbb{S}[\![\langle\mathtt{A},\mathtt{C}\rangle]\!]$, called the *concrete* semantics [10]. Set of traces are approximated are represented by suitable abstract elements, which

$$\frac{}{\langle\texttt{skip},\sigma\rangle\longrightarrow\sigma}\qquad\frac{\langle\texttt{E},\sigma\rangle\xrightarrow{\texttt{E}}\langle v,\sigma\rangle\quad v{\neq}\texttt{ExcMath}}{\langle\texttt{x=E},\sigma\rangle\longrightarrow\sigma[x{\mapsto}v]}\qquad\frac{\langle\texttt{E},\sigma\rangle\xrightarrow{\texttt{E}}\langle\texttt{ExcMath},\sigma\rangle}{\langle\texttt{x=E},\sigma\rangle\longrightarrow\langle\texttt{ExcMath},\sigma\rangle}$$

$$\frac{\langle\texttt{C}_1,\sigma\rangle\longrightarrow\sigma'}{\langle\texttt{C}_1;\texttt{C}_2,\sigma\rangle\longrightarrow\langle\texttt{C}_2,\sigma'\rangle}\qquad\frac{\langle\texttt{C}_1,\sigma\rangle\longrightarrow\langle\texttt{Exc},\sigma\rangle}{\langle\texttt{C}_1;\texttt{C}_2,\sigma\rangle\longrightarrow\langle\texttt{Exc},\sigma\rangle}$$

$$\frac{\langle\texttt{E},\sigma\rangle\xrightarrow{\texttt{E}}\langle\underline{k},\sigma\rangle\quad\underline{k}{\neq}0}{\langle\texttt{if(E!=0)C}_1\ \texttt{else}\ \texttt{C}_2,\sigma\rangle\longrightarrow\langle\texttt{C}_1,\sigma\rangle}\qquad\frac{\langle\texttt{E},\sigma\rangle\xrightarrow{\texttt{E}}\langle 0,\sigma\rangle}{\langle\texttt{if(E!=0)C}_1\ \texttt{else}\ \texttt{C}_2,\sigma\rangle\longrightarrow\langle\texttt{C}_2,\sigma\rangle}$$

$$\frac{\langle\texttt{E},\sigma\rangle\xrightarrow{\texttt{E}}\langle\texttt{ExcMath},\sigma\rangle}{\langle\texttt{if(E!=0)C}_1\ \texttt{else}\ \texttt{C}_2,\sigma\rangle\longrightarrow\langle\texttt{ExcMath},\sigma\rangle}$$

$$\frac{\langle\texttt{E},\sigma\rangle\xrightarrow{\texttt{E}}\langle\underline{k},\sigma\rangle\quad\underline{k}{\neq}0}{\langle\texttt{while(E!=0) C},\sigma\rangle\longrightarrow\langle\texttt{C;while(E!=0) C},\sigma\rangle}\qquad\frac{\langle\texttt{E},\sigma\rangle\xrightarrow{\texttt{E}}\langle 0,\sigma\rangle}{\langle\texttt{while(E!=0) C},\sigma\rangle\longrightarrow\sigma}$$

$$\frac{\langle\texttt{E},\sigma\rangle\xrightarrow{\texttt{E}}\langle\texttt{ExcMath},\sigma\rangle}{\langle\texttt{while(E!=0) C},\sigma\rangle\longrightarrow\langle\texttt{ExcMath},\sigma\rangle}$$

$$\frac{\texttt{A}.\mathit{stdout}.\mathit{add}(\sigma(\texttt{x}),\sigma(\texttt{col}))}{\langle\texttt{write(x,col)},\sigma\rangle\longrightarrow\sigma}\qquad\frac{\texttt{Exc}{\in}\texttt{Exceptions}}{\langle\texttt{throw Exc},\sigma\rangle\longrightarrow\langle\texttt{Exc},\sigma\rangle}$$

$$\frac{\langle\texttt{C}_1,\sigma\rangle\longrightarrow\sigma'}{\langle\texttt{try C}_1\ \texttt{catch(Exc) C}_2,\sigma\rangle\longrightarrow\sigma'}\qquad\frac{\langle\texttt{C}_1,\sigma\rangle\longrightarrow\langle\texttt{Exc},\sigma'\rangle}{\langle\texttt{try C}_1\ \texttt{catch(Exc) C}_2,\sigma\rangle\longrightarrow\langle\texttt{C}_2,\sigma'\rangle}$$

$$\frac{\langle\texttt{C}_1,\sigma\rangle\longrightarrow\langle\texttt{Exc}',\sigma'\rangle\quad\texttt{Exc}'{\neq}\texttt{Exc}}{\langle\texttt{try C}_1\ \texttt{catch(Exc) C}_2,\sigma\rangle\longrightarrow\langle\texttt{Exc}',\sigma'\rangle}$$

Fig. 2. The transition relations for programs

capture interesting properties while disregarding other execution properties that are out of the scope of interest. Abstract properties (or elements) belong to an *abstract domain of observables*, $\bar{\mathsf{D}}$, and they are ordered according to $\bar{\sqsubseteq}$, the abstract counterpart for logical implication. In this work we assume that $\langle\bar{\mathsf{D}},\bar{\sqsubseteq}\rangle$ is a complete lattice.

The correspondence between the concrete and the abstract semantic domains is given by a pair of monotonic functions $\langle\alpha,\gamma\rangle$. The function $\alpha\in[\mathcal{P}(\Sigma^*)\to\bar{\mathsf{D}}]$, called the abstraction function, formalizes the notion of the abstraction, and $\alpha(T)$ represents the *best* approximation in $\bar{\mathsf{D}}$ of the set of traces T (wrt the order in $\bar{\mathsf{D}}$). If $\alpha(T)\bar{\sqsubseteq}\bar{\mathsf{p}}$ then $\bar{\mathsf{p}}$ is also a correct, although less precise, abstract approximation of T. On the other hand, the function $\gamma\in[\bar{\mathsf{D}}\to\mathcal{P}(\Sigma^*)]$, called the concretization function, returns the set of traces that are captured by an abstract property $\bar{\mathsf{p}}$. The abstraction and concretization functions must satisfy the following property:

$$\forall T\in\mathcal{P}(\Sigma^*).\forall\bar{\mathsf{d}}\in\bar{\mathsf{D}}.\ \alpha(T)\ \bar{\sqsubseteq}\ \bar{\mathsf{d}}\iff T\subseteq\gamma(\bar{\mathsf{d}}),$$

in such a case, we say that $\langle\alpha,\gamma\rangle$ form a Galois connection between the concrete and the abstract domains. We write is as

$$\langle\mathcal{P}(\Sigma^*),\subseteq\rangle\ \underset{\alpha}{\overset{\gamma}{\leftrightarrows}}\ \langle\bar{\mathsf{D}},\bar{\sqsubseteq}\rangle. \tag{1}$$

The abstract semantics of a system, $\bar{\mathbb{s}}[\![\langle \mathtt{A}, \mathtt{C} \rangle]\!]$, is defined over an abstract domain that is linked to the concrete domain by a Galois connection. It must satisfy the soundness criterion, [10]:

$$\forall S_0 \subseteq \Sigma.\ \alpha(\mathbb{s}[\![\langle \mathtt{A}, \mathtt{C} \rangle]\!](S_0)) \mathrel{\bar{\sqsubseteq}} \bar{\mathbb{s}}[\![\langle \mathtt{A}, \mathtt{C} \rangle]\!](\alpha(S_0)).$$

The soundness criterion above imposes that, when the properties encoded by a given abstract domain are considered, the abstract semantics $\bar{\mathbb{s}}[\![\langle \mathtt{A}, \mathtt{C} \rangle]\!]$ captures all the behaviors of $\langle \mathtt{A}, \mathtt{C} \rangle$. As a consequence, given a specification of a system $\langle \mathtt{A}, \mathtt{C} \rangle$ expressed as an abstract property $\bar{\mathsf{p}}$, if $\bar{\mathbb{s}}[\![\langle \mathtt{A}, \mathtt{C} \rangle]\!](\alpha(S_0)) \bar{\sqsubseteq} \bar{\mathsf{p}}$, by the soundness criterion and by the transitivity of $\bar{\sqsubseteq}$, we have that

$$\alpha(\mathbb{s}[\![\langle \mathtt{A}, \mathtt{C} \rangle]\!](S_0)) \bar{\sqsubseteq} \bar{\mathsf{p}}.$$

This means that $\langle \mathtt{A}, \mathtt{C} \rangle$ respects the specification $\bar{\mathsf{p}}$.

In the following, we instantiate the abstract domain and $\bar{\mathsf{p}}$ in order to reflect non-functional requirements of systems and we show how well-known static analyses can be re-used in this enhanced context for the automatic verification of such properties.

4 Application: Non-functional Requirement Analysis

Non-functional software requirements are requirements which are not directly concerned with the specific functions delivered by the system [35]. They may relate to emergent system properties such as reliability, response time and store occupancy. Alternatively, they may define constraints on the system like the data representation used in system interfaces.

The ‘IEEE-Std 830 - 1993’ [23] presents a comprehensive list of non-functional requirements. In the following we will focus on a few of such requirements, namely *portability*, *efficiency*, *robustness* and *usability*. The approach can be extended to cope with other non-functional requirements.

In this section, we show (i) how such requirements admit a rigorous formalization, unlike, e.g., what stated in [27, §8.2], (ii) how, by a suitable choice of abstract domains, existing tools can be re-used to verify such requirements, and (iii) the effectiveness of the approach on a public-domain static analyzer [8].

4.1 Portability

Informal Definition. According to [17], a software “*is portable if it can run on different environments*”. The term *environment* may refer to a hardware platform or a software environment. Analogously, another widespread textbook, [31], defines portability as “*the ease of transferring software products to various hardware and software environments*”. The first observation is that the two definitions implicitly link the requirement to unspecified software metrics. Furthermore, as any natural-based language specifications, they are intrinsically ambiguous. For instance, the word “*run*” can be read as just the possibility of recompiling and executing the software on different system, but also as the request that some behavioral properties of the software are preserved in different platforms.

Formal Definition. We specify portability as a property of the execution of a program that is preserved when it is ported on different architectures. This means that up to a certain property of interest, the behavior of a software is the same on a different architecture.

Definition 1 (Portability). *Let us consider a program* $\mathtt{C}$*, an architecture* $\mathtt{A}$ *and a Galois connection* $\langle \mathcal{P}(\Sigma^*), \subseteq \rangle \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \langle \bar{\mathsf{D}}, \bar{\sqsubseteq} \rangle$*. We say that* $\mathtt{C}$*, developed on* $\mathtt{A}$*, is portable on the architecture* $\mathtt{B}$ *w.r.t. the observable domain* $\bar{\mathsf{D}}$*, if*

$$\forall S_0 \subseteq \Sigma.\ \alpha(\mathbb{S}[\![\langle \mathtt{B}, \mathtt{C} \rangle]\!](S_0)) \mathrel{\bar{\sqsubseteq}} \alpha(\mathbb{S}[\![\langle \mathtt{A}, \mathtt{C} \rangle]\!](S_0)).$$

Abstraction. A class property one is interested to keep unchanged among different porting of the software is the behavior w.r.t. arithmetic overflow. For instance, the violation of such a property in porting the control software on a different architecture was at the origin of the Arianne V crash [29].

Arithmetic overflow can be checked by using numerical abstract domains, e.g., [10,14,32]. In such domains the range of the values assumed by a variable can be constrained so that it can be checked against the largest representable number in a given architecture.

Example 1 (Portability). Let us consider the program $\mathtt{C}$ in Fig. 3(a), and let us consider an architecture $\mathtt{A}$ such that $\mathtt{A}.\mathit{bits} = 32$. We can use the Intervals abstract domain [10], and the public-domain static analyzer [8] to infer that $\bar{\mathbb{S}}[\![\langle \mathtt{A}, \mathtt{C} \rangle]\!](\mathtt{i} \mapsto [-\infty, +\infty]) = [1, 2^{16}]$, and as 2^{16} is representable on a 32 bit architecture, then program $\mathtt{C}$ does not cause any arithmetic overflow. As a consequence, by the soundness of the static analysis (guaranteed by abstract interpretation theory), we can safely infer that the program is portable to any architecture in which 2^{16} is representable (this is not the case in a 16 bits architecture).

4.2 Efficiency

Informal Definition. In the existing literature, efficiency "*refers to how economically the software utilizes the resources of the computer*" [17], or it is "*the ability of a software system to place as few demands as possible on hardware resources, such as processor time or space occupied*" [31]. Once again, such definitions suffer from the ambiguity of the natural language,e.g., it is not clear if when verifying efficiency requirements the underlying architecture must be considered or not, or if space and time requirements must be considered independently or not.

Formal Definition. Efficiency can be formally defined as an abstraction of the execution traces of a program. As such behavior depends on the underlying architecture, our definition explicitly mentions the architecture in which the program is executed. Efficiency requirements can be specified by natural numbers, standing, for instance, for the number of processor cycles or the size of the heap. As a

```
C ≜ i = 1;
    while (2^16-i != 0)
        i = i*4
```

(a) C, a program not portable on 16 bits architectures

```
C' ≜ i = 1;
     while (2^16-i != 0)
         i = i+2
```

(b) C', a non-efficient program

```
D ≜ try
        i = ?;
        if(i !=0) c = i / 0
        else throw Err
    catch(Err)
        c = 0;
    write(c,255)
```

(c) D, a robust program

```
E ≜ x = ?; r = ?; g = ?; b = ?
    if(r+g-1!=0)
      col = 2^r + 2^g + 2^b
     else col = 0;
    write(x,col)
```

(d) E, a program usable by daltonians

Fig. 3. Four programs on which we verify non-functional prequirements

consequence our abstract domain will be set of natural numbers with the usual total order, $\langle \mathbb{N}, \leq \rangle$.

We distinguish between efficiency in time and space. The first one corresponds to the length of a trace, i.e. the number of transitions for executing the system, and the second one to the size of the environment, i.e. the maximum quantity of memory allocated during program execution. It is worth noting that the following definitions are well-formed as we consider partial execution traces, i.e., (possible infinite) sets of finite traces. Recall that Ω denotes an uninitialized variable.

Definition 2 (Time Efficiency). *Let* C *be a program,* A *an architecture,* length $\in [\mathcal{P}(\Sigma^*) \to \mathbb{N}]$ *be the length of a trace, and* $\langle \mathcal{P}(\Sigma^*), \subseteq \rangle \xleftrightarrows[\alpha_t]{\gamma_t} \langle \mathbb{N}, \leq \rangle$ *a Galois connection where*

$$\alpha_t = \lambda T.\ \sup(\{\text{length}(\tau) \mid \tau \in T\})$$
$$\gamma_t = \lambda n.\ \{\tau \in \mathcal{P}(\Sigma^*) \mid \text{length}(\tau) \leq n\}.$$

We say that the system $\langle \mathtt{A}, \mathtt{C} \rangle$ *respects the time requirement* k *if*

$$\forall S_0 \subseteq \Sigma.\ \alpha_t(\mathbb{S}[\![\langle \mathtt{A}, \mathtt{C} \rangle]\!](S_0)) \leq k.$$

Definition 3 (Space Efficiency). *Let* C *be a program,* A *an architecture,* size $\in [\mathcal{P}(\Sigma) \to \mathbb{N}]$ *be the function defined as*

$$\text{size} = \lambda\sigma.\ \#\{\mathtt{x} \in \mathtt{Vars} \mid \sigma(\mathtt{x}) \neq \Omega\},$$

and $\langle \mathcal{P}(\Sigma^*), \subseteq \rangle \xleftrightarrows[\alpha_s]{\gamma_s} \langle \mathbb{N}, \leq \rangle$ *a Galois connection where*

$$\alpha_s = \lambda T.\ \max_{\tau \in T}\{\text{size}(\sigma) \mid \sigma \in \tau\}$$
$$\gamma_s = \lambda n.\ \{\tau \in \mathcal{P}(\Sigma^*) \mid \forall \sigma \in \tau.\ \text{size}(\sigma) \leq n\}$$

We say that the system $\langle \mathtt{A}, \mathtt{C} \rangle$ *respects the space requirement* k *if*

$$\forall S_0 \subseteq \Sigma.\ \alpha_s(\mathbb{S}[\![\langle \mathtt{A}, \mathtt{C} \rangle]\!](S_0)) \leq k.$$

Abstractions. In order to automatically verify time requirements, we must find an upper bound to the number of transitions performed during the execution of a system. Once again, we can do it by using a numerical abstract domain. In fact, we can endow a concrete state σ with a (hidden) variable `time`, to be incremented at each transition [18]. Then, the values taken by `time` will be upper-approximated in the numerical domain, say by $\overline{\mathtt{time}}$, so that the verification boils to check that $\overline{\mathtt{time}} \leq k$. In the same way, the verification of space requirements can be obtained by abstracting a state with the number of variables different from Ω it contains. The approach can be generalized to more complex languages, e.g., a language with recursive functions. In this case, the stack will be approximated by its height.

In our approach, verification of time and space efficiency requirements can be easily combined by considering the reduced product of the two abstract domains [10].

Example 2 (Efficiency). Let us consider the programs $\mathtt{C}$ and $\mathtt{C}'$ in Fig. 3, an architecture $\mathtt{A}$, where the multiplication is a primitive operation, and an architecture $\mathtt{A}'$ where the multiplication is implemented as a sequence of additions, e.g., $\mathtt{i} = \mathtt{i} * 4$ becomes $\mathtt{i} = \mathtt{i} + \mathtt{i}; \mathtt{i} = \mathtt{i} + \mathtt{i}$. Using the analyzer described in [8], we can infer:

$$\begin{aligned}
&\bar{\mathbb{S}}[\![\langle \mathtt{A}, \mathtt{C} \rangle]\!](\langle \mathtt{i} \mapsto [-\infty, +\infty], \mathtt{time} \mapsto \underline{0} \rangle) = \langle \mathtt{i} \mapsto [1, 2^{16}], \mathtt{time} \mapsto [0, 9] \rangle \\
&\bar{\mathbb{S}}[\![\langle \mathtt{A}', \mathtt{C} \rangle]\!](\langle \mathtt{i} \mapsto [-\infty, +\infty], \mathtt{time} \mapsto \underline{0} \rangle) = \langle \mathtt{i} \mapsto [1, 2^{16}], \mathtt{time} \mapsto [0, 25] \rangle, \\
&\bar{\mathbb{S}}[\![\langle \mathtt{A}, \mathtt{C}' \rangle]\!](\langle \mathtt{i} \mapsto [-\infty, +\infty], \mathtt{time} \mapsto \underline{0} \rangle) = \langle \mathtt{i} \mapsto [0, 2^{16}], \mathtt{time} \mapsto [0, 32769] \rangle.
\end{aligned}$$

Observe that the results above can be used for comparing different programs on different architectures.

4.3 Robustness

Informal Definition. Robustness, or dependability, for [17] is "*the ability of a program to behave reasonably, even in circumstances that were not anticipated in the specifications*", for [31] is "*the ability of software systems to react appropriately to abnormal conditions*", and for [27] is "*the time to restart after failure*". Once again, the three definitions are not rigorous enough: the first definition does not specify what is a reasonable behavior, the second one does not specify what is an abnormal condition, and the latter has implicit the strong assumption that all possible failures are considered.

Formal Definition. A software is robust, if any exception raised during its execution, in any architecture and with any initial state, is caught by some exception handler. We recall that exceptions can be raised either by the architecture, e.g., division-by-zero, or by the software itself. As a consequence, a robust program never terminates in an exceptional state.

Definition 4 (Robustness). *Let* C *be a program, and let* $\langle \mathcal{P}(\Sigma^*), \subseteq \rangle \xrightleftharpoons[\alpha_d]{\gamma_d} \langle \mathcal{P}(\Sigma), \subseteq \rangle$ *be a Galois connection where*

$$\begin{aligned} \alpha_d &= \lambda T.\ \{\sigma_n \mid \sigma_0 \ldots \sigma_n \in T\} \\ \gamma_d &= \lambda S.\ \{\sigma_0 \ldots \sigma_{n-1}\sigma_n \mid \forall i \in [0, n-1].\sigma \in \Sigma \wedge \sigma_n \in S\}. \end{aligned}$$

We say that a system is robust if for all the architectures A*,*

$$\forall S_0 \in \mathcal{P}(\Sigma).\ \alpha_d(\$[\![\langle \mathtt{A}, \mathtt{C}\rangle]\!](S_0)) \cap \mathtt{Exceptions} \times \mathsf{Env} = \emptyset.$$

Abstraction. Robustness can be checked either by considering an abstract domain for inferring the uncaught exceptions [34], or by considering an abstract domain for reachability analysis [8]. In the first case, a program is robust if the analysis reports that no exception can be raised; in the latter, a program is robust if the analysis reports that the lines of code that may raise an exception (e.g., with a `throw` statement) are never reached.

Example 3 (Robustness). Let us consider the program D of Fig. 3(c). An interval analysis determines that when the true-branch of the `if` statement is taken, `i` is different from zero, so that the `MathErr` exception cannot be raised. In the other case, the exception `Err` is raised and then it is also caught. As a consequence, D is robust w.r.t. the chosen abstraction.

4.4 Usability

Informal Definition. The definition of usability is probably the most contrived one. The definition in [17] says that "*software system is usable [...] if its human users find it easy to use*", whereas [31] talks about ease of use as "*the ease with which people of various backgrounds [...] can learn to use software*" and [27] defines it in function of other, undefined, basic concepts as "*learnability, satisfaction, memorability*".

Formal Definition. In our setting, usability is a abstraction of the output stream that is preserved when a given property, depending on the particular user, is considered. For instance, an abstraction that considers the colors of the output characters can be used to verify if a system is usable for daltonians. We need some auxiliary definitions. Output streams belong to the set `Stdout`. Given a state $\sigma \in \Sigma$, the function out $\in [\Sigma \to \mathtt{Stdout}]$ is such that $\mathrm{out}(\sigma)$ is the output stream in the state σ.

Definition 5 (Usability). *Let* C *be a program,* A *an architecture, let* $\langle \mathcal{P}(\Sigma^*), \subseteq \rangle \xrightleftharpoons[\alpha_\Sigma]{\gamma_\Sigma} \langle \mathcal{P}(\mathrm{Stdout}), \subseteq \rangle$ *be a Galois connection where*

$$\begin{aligned} \alpha_\Sigma &= \lambda T.\ \{\mathrm{out}(\sigma) \in \Sigma \mid \exists \tau \in T.\ \sigma \in T\} \\ \gamma_\Sigma &= \lambda O.\ \{\tau \in \Sigma^* \mid \forall \sigma \in \tau.\ \exists o \in O.\ \mathrm{out}(\sigma) = o\}, \end{aligned}$$

let $\langle \mathcal{P}(\mathrm{Stdout}), \subseteq \rangle \xrightleftharpoons[\alpha]{\gamma} \langle \bar{\mathsf{D}}, \bar{\sqsubseteq} \rangle$ *be a Galois connection, and let* $\bar{\mathsf{p}} \in \bar{\mathsf{D}}$*. We say that the system* $\langle \mathtt{A}, \mathtt{C} \rangle$ *is usable w.r.t. the observable* $\bar{\mathsf{p}}$ *if*

$$\forall S_0.\ \alpha(\alpha_\Sigma(\$[\![\langle \mathtt{A}, \mathtt{C}\rangle]\!])(S_0)) \mathrel{\bar{\sqsubseteq}} \bar{\mathsf{p}}.$$

Abstract Domains. The definition above can be instantiated to consider the usability of a system for daltonians, i.e., people afflicted by red/green color blindness. In fact, the colors of the output stream can be abstracted in order to collapse together colors indistinguishable by daltonians. As colors are represented by integers in the RGB color system, numerical abstract domains can be used to automatically check properties on colors.

Example 4 (Usability). Let us consider the program $\mathtt{E}$ in Fig. 3(d), an architecture where the input stream is a sequence of 0/1 digits, and colors are represented as in RGB schema using 3 bits, i.e. colors range between 0 (black) and 7 (white). Using the static analyzer of [8] instantiated with the Intervals abstract domain, and refined with trace partitioning [21], one infers that

$$\begin{aligned}\bar{\mathbb{S}}[\![\langle\mathtt{A},\mathtt{E}\rangle]\!](\langle \mathtt{x}\mapsto[0,1],\mathtt{r},\mathtt{g},\mathtt{b}\mapsto[0,1]\rangle)\\ =(\langle\mathtt{x}\mapsto[0,1],\mathtt{r},\mathtt{g},\mathtt{b}\mapsto[0,1],\mathtt{col}\mapsto[0,1]\cup[6,7]\rangle),\end{aligned}$$

so that as $\mathtt{col}$ is always in the set of the colors distinguishable by daltonians (i.e. { black, blue, yellow, white}), $\mathtt{E}$ respects the usability specification.

4.5 Other Non-functional Requirements

We showed how four typical non-functional requirements can be encapsulated in our framework. This approach based on preservation of a property up to a given observation, can be easily generalized to other product non-functional requirements. For instance, *upgrade* means that when a new program $\mathtt{N}$, replaces a program $\mathtt{O}$ on a given architecture $\mathtt{A}$, then the observed behavior is preserved: $\alpha(\mathbb{S}[\![\langle\mathtt{A},\mathtt{N}\rangle]\!])\bar{\sqsubseteq}\alpha(\mathbb{S}[\![\langle\mathtt{A},\mathtt{O}\rangle]\!])$. Similarly, if *compatibility* is a property specified by an abstract element $\bar{\mathtt{c}}$, then we say that two programs $\mathtt{P}$ and $\mathtt{P}'$ are compatible w.r.t. $\bar{\mathtt{c}}$ if $\alpha(\mathbb{S}[\![\langle\mathtt{A},\mathtt{P}\rangle]\!])\bar{\sqsubseteq}\bar{\mathtt{c}}$ and $\alpha(\mathbb{S}[\![\langle\mathtt{A},\mathtt{P}'\rangle]\!])\bar{\sqsubseteq}\bar{\mathtt{c}}$.

5 Conclusions and Future Work

In this paper, we faced the issue of applying Abstract Interpretation theory in order to model non functional software requirements and to support their automatic validation.

Recent very encouraging experiences show that abstract interpretation-based static program analysis can be made efficient and precise enough to formally verify a class of properties for a family of large programs with few or no false alarms, also in case of critical embedded systems [3]. We strongly believe that also the treatment of non functional requirements can well fit in this picture.

Two research directions seem particularly promising, in this respect: (i) the design of a library of sophisticated abstract domains for non functional requirements validation, and (ii) the automatic derivation of metrics associated to the domain of observables. In the first case, as observed in Example 2, the precision of the analysis can be greatly improved by a suitable choice of operators on domains (e.g., reduced product [11], and open product [6]). In the second case, it would be interesting to study *abstract metrics*, i.e. metrics tunable with respect to a given observable. In fact, any (even infinite) finite-height domain of observables can be associated with at least two metrics, by considering as distance between two abstract properties

$$\rho_1(d_1, d_2) = \min\{\text{length}(d_i, d_1 \sqcup d_2) \mid i \in \{1, 2\}\}$$
$$\rho_2(d_1, d_2) = \min\{\text{length}(d_i, d_1 \sqcap d_2) \mid i \in \{1, 2\}\}$$

where *length* returns the length of the path in the domain $\bar{\mathsf{D}}$. ρ_1 computes the lack of precision with respect to the element that represents the union of the information the two elements contain, while ρ_2 does the same with respect to the element that keeps the common information.

References

1. J.-R. Abrial. *The B-Book: Assigning Programs to Meanings.* Cambridge University Press, 1996.
2. J.-R. Abrial. B# : Toward a synthesis between Z and B. In Didier Bert, Jonathan P. Bowen, S. King, and M. Waldn, editors, *ZB'2003 – Formal Specification and Development in Z and B*, volume 2651 of *Lecture Notes in Computer Science (Springer-Verlag)*, pages 168–177, Turku, Finland, June 2003. Springer.
3. B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. A static analyzer for large safety-critical software. In *Proceedings of the 2003 ACM Conference on Programming Language Design and Implementation (PLDI'03)*, pages 196–207. ACM Press, June 2003.
4. B. W. Boehm. Software engineering. *IEEE Transactions on Computers*, pages 1266–41, December 1976.
5. J. Brauburger. Automatic termination analysis for partial functions using polynomial orderings. *Lecture Notes in Computer Science*, 1302:330–344, 1997. In P. Van Hentenryck, editor, Proc.4 th Int. Symp.SAS 97, Paris.
6. A. Cortesi, B. Le Charlier, and P. van Hentenryck. Combinations of abstract domains for logic programming. In *21th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages POPL'94, ACM-Press, New York*, pages 227–239, 1994.
7. P. Cousot. Types as abstract interpretations, invited paper. In *24th ACM Symposium on Principles of Programming Languages (POPL '97)*, pages 316–331. ACM Press, January 1997.
8. P. Cousot. The calculational design of a generic abstract interpreter. In M. Broy and R. Steinbrüggen, editors, *Calculational System Design.* NATO ASI Series F. IOS Press, Amsterdam, 1999.
9. P. Cousot. Abstract interpretation based formal methods and future challenges, invited paper. In R. Wilhelm, editor, *Informatics — 10 Years Back, 10 Years Ahead*, volume 2000 of *Lecture Notes in Computer Science*, pages 138–156. Springer-Verlag, 2001.
10. P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *4th ACM Symposium on Principles of Programming Languages (POPL '77)*, pages 238–252. ACM Press, January 1977.

11. P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *6th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '79)*, pages 269–282. ACM Press, 1979.
12. P. Cousot and R. Cousot. Systematic design of program transformation frameworks by abstract interpretation. In *29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '02)*, pages 178–190. ACM Press, New York, NY, January 2002.
13. P. Cousot and R. Cousot. An abstract interpretation-based framework for software watermarking. In *Conference Record of the Thirtyfirst Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 173–185, Venice, Italy, January 14-16 2004. ACM Press, New York, NY.
14. P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *5th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '78)*, pages 84–97. ACM Press, 1978.
15. A. Davis. *Software Requirements: Objects, Functions and States.* Prentice Hall, 1992.
16. M. S. Deutsch and R. R. Willis. *Software Quality Engineering.* Prentice-Hall, 1988.
17. C. Ghezzi, Jazayeri M., and D. Mandrioli. *Foundamentals of Software Engineering.* Prentice Hall, 2nd edition, 2003.
18. N. Halbwachs. Determination automatique de relations lineaires verifees par les variables d'un programme. *These de 3eme cycle d'informatique, Universite scientifique et medicale de Grenoble*, March 1979.
19. N. Halbwachs. Non-functional requirements in the software development process. *Software Quality*, 5(4):285–294, 1995.
20. N. Halbwachs. About synchronous programming and abstract interpretation. *Science of Computer Programming*, 31(1):75–89, May 1998.
21. M. Handjieva and S. Tzolovski. Refining static analyses by trace-based partitioning using control flow. In *Proceedings of the Static Analysis Symposium (SAS '98)*, volume 1503 of *Lectures Notes in Computer Science*, pages 200–215. Springer-Verlag, 1998.
22. S. Hughes. Compile-time garbage collection for higher-order functional languages. *Journal of Logic and Computation*, 2(4):483–509, 1992.
23. IEEE. *IEEE Recommended Practice for Software Requirement Specification*, 1988.
24. J.M.Spivey. *The Z notation.* Prentice Hall, 1992.
25. N. D. Jones. Combining abstract interpretation and partial evaluation. In Pascal Van Hentenryck, editor, *Static Analysis*, volume 1302 of *Lecture Notes in Computer Science*, pages 396–405. Springer-Verlag, 1997. In P. Van Hentenryck, editor, Proc.4 th Int. Symp.SAS 97, Paris.
26. J. Karlsson. Managing software requirements using quality function deployment. *Software Quality Control*, 6(4):311–326, 1997.
27. G. Kotonya and I. Sommerville. *Requirements Engineering - Processes and Techniques.* Wiley, 1998.
28. D. Kozen, J. Palsberg, and M. I. Schwartzbach. Efficient inference of partial types. *Journal of Computer and System Sciences*, 49(2):306–324, October 1994.
29. P. Lacan, J. N. Monfort, L.V.Q. Ribal, A. Deutsch, and A. Gonthier. The software reliability verification process: The ariane 5 example. In *Proceedings DASIA 98 DAta Systems In Aerospace, Athens, GR. ESA Publications*, 1998.
30. D. Le Métayer. Program analysis for software engineering: New applications, new requirements, new tools. *ACM Computing Surveys*, 28(4es):167, December 1996.
31. B. Meyer. *Object-Oriented Software Construction.* Professional Technical Reference. Prentice Hall, 2nd edition, 1997.

32. A. Miné. The octagon abstract domain. In *AST 2001 in WCRE 2001*, IEEE, pages 310–319. IEEE CS Press, October 2001. `http://www.di.ens.fr/~mine/publi/article-mine-ast01.pdf`.
33. D. Monniaux. Abstracting cryptographic protocols with tree automata. *Science of Computer Programming*, 47(2–3):177–202, May/June 2003.
34. F. Pessaux and X. Leroy. Type-based analysis of uncaught exceptions. *ACM Transactions on Programming Languages and Systems*, 22(2):340–377, bo 2000.
35. I. Sommerville. *Software Engineering.* Addison Wesley, 6th edition, 2000.
36. K. R. Traub, D. E. Culler, and K. E. Schauser. Global analysis for partitioning non-strict programs into sequential threads. *ACM LISP Pointers*, 5(1):324–334, 1992. Proceedings of the 1992 ACM Conference on LISP and Functional Programming.