Contract Based Embedded Software Design

Christian Lidström(✉) and Dilian Gurov(✉)

KTH Royal Institute of Technology, Stockholm, Sweden
{clid,dilian}@kth.se

Abstract. In embedded systems development, *contract based design* is a design paradigm where a system is divided hierarchically into components and developed in a top-down manner, using contracts as a means to divide responsibilities and manage the complexity of the system. Contract theories provide a formal basis for reasoning about the properties of the system, and of the contracts themselves. In previous work, we have developed a contract theory for sequential, procedural programs, that is defined abstractly, at the semantic level. The theory fulfils well-established, desired properties of system design. In this paper, we present a methodology for applying the contract theory in real embedded software design. We show how to instantiate the contract theory with concrete syntaxes for defining components and contracts, and how the contract theory enables formal reasoning about the resulting objects. In order to cope with the need for different behavioural models at different levels of abstraction in an embedded system, we extend the contract theory through parametrisation on the semantic domain. We illustrate the application of the proposed methodology on a small, but realistic example, where the temporal logic TLA is used for reasoning at the system level, while lower level components are specified using pre- and post-conditions in the form of ACSL, a specification language for C.

1 Introduction

A *contract* for a software component is a means to specify the behaviour (or result) the component has to produce, called the guarantee, provided that the user (or client) of the component fulfils certain constraints on how it interacts with the component, called assumptions. Software contracts were pioneered by the works of Floyd [14] and Hoare [15]. In Hoare logic, meaning is assigned to sequential programs axiomatically, through so-called Hoare triples, allowing the desired relationship between initial and final values of certain variables to be specified. Specifying contracts in this way has been advocated by Meyer with the design methodology Design-by-Contract [23]. The methodology is well-suited for *independent implementation and verification*, meaning that development of software components occurs independently, without knowledge of any implementation details of other components, but instead relying on the contracts of the latter. Contract based design as an approach to *systems design* thus provides a way to deal with the complexity of large systems, making explicit the assumptions on each component's environment. In a top-down design flow, contracts are

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2023
C. David and M. Sun (Eds.): TASE 2023, LNCS 13931, pp. 77–94, 2023.
https://doi.org/10.1007/978-3-031-35257-7_5

iteratively decomposed into sub-contracts [7,9,17,28], and the typical task then is to show that the *composition* of the sub-contracts *refines* the original contract. Contract refinement and composition enable the *independent development* and *reuse* of components. Further, contract *conjunction* allows the superimposition of contracts over the same component, when they concern different aspects of its behaviour. *Contract theories* formalise the abstract notion of a contract [8], and define operations and relations on contracts mentioned above.

Motivation. In previous work [21], we proposed a contract theory for reasoning about procedural programs, and takes therefore *procedures* as the basic building block (i.e., component). The theory is abstract in that it is developed purely at the semantic level, in terms of standard *denotational semantics*. We showed, in the context of a simplistic imperative programming language with procedures, that Hoare logic contracts and their procedure-modular verification can be cast naturally in our contract theory. We also showed that the contract theory instantiates the meta-theory of Benveniste et al. [8], and thus fulfils certain well-established and desired properties of embedded systems development. In a follow-up work [2], we showed in practice how to combine state-of-the-art verification tools for different semantic domains, and presented a proof-of-concept tool chain for this purpose. We also sketched how the contract theory can serve as formal basis for this approach, but left out a fully formal justification.

To illustrate the usefulness of the contract theory, the present paper presents a *methodology* of how to apply it in embedded systems development. In embedded systems, the type of behaviour that is of interest typically differs at different levels in the hierarchy of the system. At lower levels, software components are often understood as *state transformers*, i.e., as components the purpose of which is to transform certain initial values to certain final values, where the intermediate values are just implementation details irrelevant to the computed function. Such programs are appropriately specified using Hoare logic contracts in the form of pre- and post-conditions. At higher levels, however, such specifications are usually not sufficient, since the behaviour depends on *interaction* between several software sub-systems. Here, specifications are typically of a *temporal* nature, and the intermediate states of the program execution cannot be ignored. Since our previous work only dealt with the former case, we will show here that our framework is also fit for reasoning about the temporal behaviour of programs. Even more, we extend the contract theory to handle the *combination* of the different notions of behaviour, as needed for embedded systems development.

Example. Embedded systems typically interact with their environment through sensors and actuators, and have the goal to maintain certain temporal properties. Say we want to design a system, which (among other things) uses a sensor to measure the temperature of a part of the system, and displays it to a user. The temperature is measured in the unit of Kelvin, but should be displayed to the user in Celsius. A desired temporal property of this system is:

At any point in time, when the temperature is read from the sensor
it must eventually be displayed, in Celsius. (1)

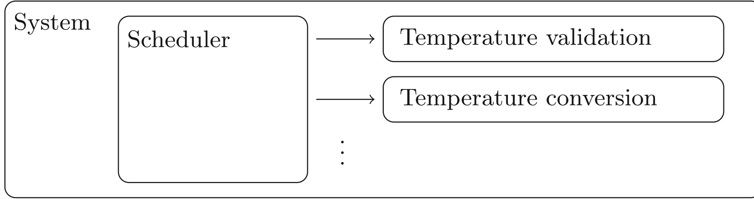

Fig. 1. Division of the example system into components

A formalised specification, or contract, of this property would be the starting point for the design of the system. Then, the system is divided into appropriate proto-components (i.e., components yet to be specified), defining the intended system architecture, and the contract is decomposed into contracts for those components, in such a way that their composition refines the top-level contract. Reasonably, such a system will have a function that reads the sensor value, and checks for potential erroneous inputs, such as in the case of potential hardware faults. The system will then have a function that converts the sanitised sensor value, and displays it (which we will model as writing to a variable). Finally, a main, *scheduling* component will continuously execute the other components. Figure 1 illustrates the system architecture, where the arrows indicate procedure calls between components. As pointed out, the top-level property is of a temporal nature, whereas the two sub-components perform computations of a transformational nature. This has to be reflected by their respective contracts.

Contributions. We present a *methodology* for designing embedded software, formally underpinned by our contract theory [21]. The contract theory provides a formal basis for system design as per the principles of contract based design. The contract theory is *abstract*, allowing instantiations with different specification and implementation languages, and is based on *procedures*, making it fit for software development. To allow the combination of different execution domains, we generalise here the contract theory of [21] through *parametrisation* on the domain of execution behaviour of system components.

We show how to instantiate the parametrised contract theory using, in combination, two semantic domains of executions: state pairs and infinite traces. We use established languages and tools to support this instantiation, such as the Frama-C [18] (whose specification language is ACSL [4]) and TLA [19,20] frameworks. We show how high-level, temporal contracts expressed in TLA can be decomposed into lower-level, state-transformation contracts expressed in ACSL. This enables procedure-modular verification of embedded software implemented in C, relative to procedure contracts, while satisfying the high-level contracts. The whole system design process is exemplified on industrial-like software.

Structure. The paper is organised as follows. Section 2 introduces the concepts of contract based design, contract theories and meta-theories. Section 3 presents our abstract, and parametrised, contract theory. Section 4 describes abstractly

how the contract theory can be applied, and what is required to instantiate it. Section 5 shows how the contract theory can be instantiated with concrete languages for specifying contracts, in two different domains. Section 6 shows how components can be concretely defined (that is, implemented) using the C language, and describes procedure-modular verification in the context of the two domains. Section 7 illustrates the application of the contract theory on our running example. Section 8 details the related work. We conclude with Sect. 9.

2 Contract Based Design and Contract Theories

Contract based design [8,23] is a systems design paradigm where the design of a system is performed in a top-down manner, through the use of contracts. Top-level properties are specified as a contract for the system as a whole. The system is then divided, conceptually, into *components* intended to perform specific tasks, defining the system architecture. The top-level contract is *decomposed* into contracts for the components of the system, documenting their intended behaviour. Base components (i.e., components which are not further divided) are then implemented individually and independently, by relying on the contracts of other components. Finally, the components are assembled iteratively, in a bottom-up manner, until the whole system is formed. Contracts must thus expose enough information to the other components about the provided functionality, and state the expectations on the rest of the system.

A common design pattern are *assume-guarantee* contracts. Here, a contract consists of a set of *assumptions* and a set of *guarantees*. The assumptions specify how the component expects the system, or environment, to behave. For example, it may require that certain other components exist, and that they can be interfaced with in a specific way. The guarantees specify how the component obliges itself to behave, e.g., how it can be interfaced with, and what computations it performs, under the condition that the assumptions are adhered to.

Our focus here is on the design of embedded software, where the end-goal is to fulfil some system-wide properties. Thus, as contracts are decomposed and components are implemented to satisfy them, it is important that this is performed so that, when the final system is assembled, the top-level properties hold. Benveniste et al. [8] examined existing system design methodologies, and identified two properties as essential for any design framework: *independent implementation*, and *reuse*, of components. To this end, one should be able to *decompose* and *compose* components as well as contracts. In addition, the framework must also allow for the *refinement* and *abstraction* of contracts, in order to expose only the information appropriate at specific levels of abstraction.

Contract Theories and Meta-Theories. Contract theories provide formal frameworks for contract based system design, defining the basic units of reasoning, and the operations and relations over them. In turn, contract meta-theories systematise such contract theories by axiomatising the desired properties of their relations and operations, instead of defining them explicitly. Benveniste et al. [8]

develop such a meta-theory, which systematises contract theories for system design in cyber-physical systems. As such, it states that a contract theory must have a notion of *component* and a notion of *contract*. Further, there must be *composition* operators over both notions, a binary *refinement* relation between contracts, and a *conjunction* operator over contracts. While their definitions are left abstract, the meta-theory formulates axioms that must be fulfilled by contract theories in order to enable proper design-chain management. For instance, one axiom stated by the meta-theory postulates that when a contract $\mathcal{C}_1$ refines another contract $\mathcal{C}_2$, written $\mathcal{C}_1 \preceq \mathcal{C}_2$, then every implementation of $\mathcal{C}_1$ must also implement $\mathcal{C}_2$. This ensures that contracts can be extracted and implemented at the appropriate level of abstraction, while not affecting the higher-level contracts. For a comprehensive view of the axioms, see the original monograph [8], or the paper in which our contract theory was originally presented [21]. In Sect. 3, we present an extended version of our contract theory, in which the semantic domain of component behaviour is left as a parameter.

3 An Abstract Parametrised Contract Theory

We have previously proposed an abstract contract theory for procedural languages [21], based on a denotational semantics over the domain $\mathbf{State} \times \mathbf{State}$, where $\mathbf{State}$ denotes the set of states. The contract theory was solely defined at the semantic level, without proposing any concrete languages for defining contracts and components. In this paper, we generalise the contract theory by parametrising the semantic domain. This enables the use of arbitrary domains, allowing a wider range of concrete syntaxes, and, in turn, types of properties to be specified. As we shall see, the new contract theory supports also the *combination* of several domains, depending on their relevance for the respective component. As in the original work, it supports the design-by-contract methodology developed by Meyer [23], and satisfies the axioms of the meta-theory of Benveniste et al. [8]. Thus, the contract theory satisfies important properties desired in system design methodologies, such as *independent development*, and *reuse*, of components. This section summarises the generalised abstract contract theory, leaving out the technical details not needed to follow this paper. The full definition of the contract theory can be found in [2].

We focus on procedural languages, and assume a finite set of procedure names $\mathcal{P}$. We consider an abstract notion of behaviour, called a *run*, representing a single execution of a system. Let $\mathbf{Run}$ denote the set of all runs. For any set of procedure names $P \subseteq \mathcal{P}$, a *procedure environment* $\mathbf{Env}_P = P \rightarrow 2^{\mathbf{Run}}$ maps procedure names to corresponding sets of runs. We define a partial order relation on procedure environments as point-wise set inclusion: for any $\rho \in \mathbf{Env}_P$ and $\rho' \in \mathbf{Env}_{P'}$, $\rho \sqsubseteq \rho'$ iff $P \subseteq P'$ and $\forall p \in P. \rho(p) \subseteq \rho'(p)$. Let $\mathbf{Env} = \bigcup_{P \subseteq \mathcal{P}} \mathbf{Env}_P$. Then, $(\mathbf{Env}, \sqsubseteq)$ is a complete lattice, since for every subset of $\mathbf{Env}$ there exists a greatest lower bound (*glb*) and a least upper bound (*lub*). The respective *glb* and *lub* operations on environments are denoted by $\sqcap$ and $\sqcup$. For two environments $\rho_1 \in \mathbf{Env}_{P_1}$ and $\rho_2 \in \mathbf{Env}_{P_2}$, $\rho_1 \sqcap \rho_2$ is the environment $\rho \in \mathbf{Env}_{P_1 \cap P_2}$ such that $\forall p \in P_1 \cap .\rho(p) = \rho_1(p) \cap \rho_2(p)$.

An *interface* $I = (P^-, P^+)$ is a pair of disjoint sets of procedure names. Both *components* and *contracts* are equipped with interfaces. P^+ are the procedures *provided* by a component, i.e., procedures that are (or will be) implemented within it. Conversely, P^- are the procedures *required* by the component, i.e., those called from it but not implemented in it. A *component* m with interface $I_m = (P_m^-, P_m^+)$ is a monotonic mapping of type $m : \mathbf{Env}_{P_m^-} \rightarrow \mathbf{Env}_{P_m^+}$, i.e., a function giving the behaviour of provided procedures, depending on the behaviour of required procedures. Two components m_1 and m_2 are *composable* if and only if $P_{m_1}^+ \cap P_{m_2}^+ = \varnothing$, and the (sequential) composition is defined in terms of fixed-points. A *denotational contract* c with interface $I_c = (P_c^-, P_c^+)$ is a pair (ρ_c^-, ρ_c^+), where the required procedure environment $\rho_c^- \in \mathbf{Env}_{P_c^-}$ and the provided procedure environment $\rho_c^+ \in \mathbf{Env}_{P_c^+}$. The rationale behind these definitions is that a component m, when given the environment ρ_c^-, implements the contract, denoted $m \models c$, if the resulting procedure environment is at least as strict as ρ_c^+, under some additional restrictions on their interfaces. Furthermore, contract c *refines* contract c', denoted $c \preceq c'$, if and only if $\rho_{c'}^- \sqsubseteq \rho_c^-$ and $\rho_c^+ \sqsubseteq \rho_{c'}^+$. Contract composition has similar restrictions on the interfaces as for components, and for every procedure required by one contract and provided by the other, the provided behaviour must be at least as strict as the required one. Then, the *composition* of two composable contracts $c_1 = (\rho_{c_1}^-, \rho_{c_1}^+)$ and $c_2 = (\rho_{c_2}^-, \rho_{c_2}^+)$ with interfaces $I_{c_1} = (P_{c_1}^-, P_{c_1}^+)$ and $I_{c_2} = (P_{c_2}^-, P_{c_2}^+)$ is the contract $c_1 \otimes c_2 \stackrel{\text{def}}{=} (\rho_{c_1 \otimes c_2}^-, \rho_{c_1}^+ \sqcup \rho_{c_2}^+)$, where $\rho_{c_1 \otimes c_2}^-$ is the glb of $\rho_{c_1}^-$ and $\rho_{c_2}^-$, but restricted to those procedures not provided by one of the contracts.

The following two theorems correspond to Theorem 1 and 2 in [21]. Since the original proofs do not rely on any particular domain of runs, they apply also here, and are therefore omitted.

Theorem 1. *Component composition is well-defined, i.e., the involved fixed-points exist, and the resulting components are monotonic mappings.*

Note that for this result to hold, when instantiating with a concrete semantics, one must restrict the base components to be monotonic.

The next result captures the essential properties of composition.

Theorem 2. *For any composable contracts c_1 and c_2, and any implementations $m_1 \models c_1$ and $m_2 \models c_2$, m_1 and m_2 are composable. Furthermore, $c_1 \otimes c_2$ is the least contract w.r.t. the refinement order for which the following holds:*

(i) $m_1 \times m_2 \models c_1 \otimes c_2$, *and*
(ii) *if m is an environment to $c_1 \otimes c_2$, then $m_1 \times m$ is an environment to c_2 and $m \times m_2$ is an environment to c_1.*

4 The Contract Theory as Basis for System Design

This section describes how the abstract contract theory can be applied for the design of, and reasoning about, real systems. By *applying* the contract theory,

we mean casting the concrete specification and implementation languages in its framework, thus showing that the desired properties of system design methodologies hold. We first describe how the parametrised contract theory can be instantiated with concrete languages, possibly using several semantic domains, and then discuss procedure-modular verification in this setting.

4.1 Instantiating the Contract Theory

Instantiating the parametrised contract theory requires certain steps to be taken. Concretely, the following *elements* must be provided:

1. A *semantic domain* for runs.
2. A syntax for defining components, i.e., a *programming language*.
3. A syntax for defining contracts, i.e., a *specification language*.
4. A *contract-relative* semantics mapping concretely defined components and contracts to abstract ones, as denotations over the chosen semantic domain, under certain conditions.

The last item requires some explanation. We take the view of contracts as being separate from their implementation, and desire a program S to satisfy its contract C, denoted $S \models C$, precisely when $[\![S]\!] \subseteq [\![C]\!]$, where $[\![C]\!] \in 2^{\mathbf{Run}}$ is some given denotational semantics of C. Equivalently, one can require that the denotational semantics for contracts be defined so that the following holds:

$$[\![C]\!] = \bigcup_{S \models C} [\![S]\!] \tag{2}$$

Following the design-by-contract methodology, the semantics of procedures should be given *relative to the contracts* of the other procedures. Assuming that every procedure p is equipped with a contract C_p, we introduce the *contract environment* ρ_c, defined by $\rho_c(p) \stackrel{\text{def}}{=} [\![C_p]\!]$ for all $p \in P$, and use it to define a contract-relative semantics $[\![S]\!]^{cr} \stackrel{\text{def}}{=} [\![S]\!]_{\rho_c}$. In particular, the denotation of procedure calls is obtained from this environment (and not from solving fixed-point equations). This semantics naturally induces a contract-relative satisfaction relation $S_p \models^{cr} C_p$, where S_p is a procedure-body and C_p its contract. The contract-relative semantics must fulfil the following *properties*:

1. All base components must be monotonic mappings.
2. For any two disjoint sets of functions P_1^+ and P_2^+, abstracted individually into components m_1 and m_2, respectively, and $P_1^+ \cup P_2^+$ abstracted into component m, we must have $m_1 \times m_2 = m$.
3. For any procedure p with procedure contract C_p, abstracted into component m_p with contract c_p, we must have $S_p \models^{cr} C_p$ whenever $m_p \models c_p$.

The first restriction is needed for Theorem 1 to hold. The second restriction establishes, together with the commutativity and associativity of component composition, that the order in which we choose to abstract and compose components is unimportant. The third restriction establishes that contract satisfaction at the concrete level is consistent with that at the abstract level. The latter two restrictions enable *procedure-modular verification*, which we now discuss.

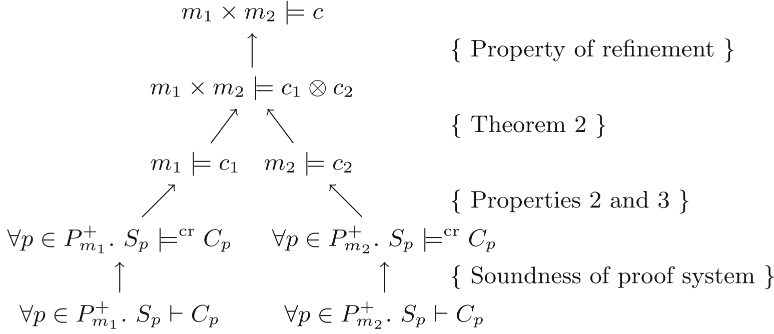

Fig. 2. View of procedure-modular verification, assuming $c_1 \otimes c_2 \preceq c$

4.2 Procedure-Modular Verification

The contract-relative semantics provides a basis for procedure-modular verification, which is performed at the syntactic level. Figure 2 illustrates how procedure-modular verification of concrete implementations against concrete specifications enables us to show, in the end, that the composition of the parts of a system fulfils some higher-level contract. We shall explain the scheme in a top-down manner. Given that $c_1 \otimes c_2 \preceq c$ has already been established (i.e., the top-level contract c has been refined and decomposed), we obtain from the properties of refinement within the contract-theory, and Theorem 2, that for the final result to hold, each sub-component must be shown to implement its contract. The properties listed in Sect. 4.1 entail that this can be shown by establishing contract satisfaction at the concrete level, which in turn follows from (syntactic) verification, assuming that the latter is sound w.r.t. the contract-relative semantics. Thus, the problem of proving high-level properties in a system composed of many, independently implemented components, is reduced to procedure-modular verification (at the syntactic level), and to showing refinement and decomposition of contracts. Syntactic, procedure-modular verification is supported by a variety of existing frameworks such as Frama-C [18] and OpenJML [13]. In the next sections, we will see concretely how, by treating contracts in one domain as a subset of another domain, we can procedure-modularly verify contracts expressing temporal properties relative to transformational contracts.

5 Specifying and Decomposing Contracts

Following the principles of contract based design outlined in Sect. 2, a system is designed in a top-down manner. Many formal languages for specifying contracts already exist, and this section describes how two such languages can be used to instantiate the contract theory. We chose, at the higher hierarchical level, to instantiate it with TLA [20], a well-known high level language for temporal reasoning about systems. On the lower level, we chose ACSL [4], the specification language of Frama-C [18], a mature tool for verification of software

written in C, one of the programming languages most commonly used in embedded systems. Frama-C includes plugins for deductive verification of C functions against contracts [5]. The two languages have different semantic domains, and are thus appropriate for specifying different aspects of the system. We show how to combine the two domains within the contract theory, and how to decompose contracts written in one language to contracts written in both.

5.1 Specifying Temporal Contracts with TLA

In embedded systems development, the top-level properties that are of greatest interest are typically of temporal nature, such as various safety and liveness properties. For this reason, we consider the domain of infinite traces (i.e., sequences of states) $\sigma = s_1, s_2, \ldots$, over states $s_i \in \mathbf{State}$, which we still leave abstract, but for which we assume that in a particular state we know the values of the variables. We denote the set of all traces by $\mathbf{State}^\omega$.

Temporal Logic of Actions. (TLA) [19,20] is a logic for reasoning about the temporal behaviour of (usually concurrent) systems. Central to TLA is the notion of *action*, which is a state predicate containing variables, constants, and logical operators. Variables can be either so-called *flexible* or *rigid* variables. Flexible variables represent the program variables, whereas rigid variables are constant but unknown values, and can be used to express relations between states. Rigid variables are implicitly universally quantified. Actions are evaluated over pairs of states, representing a (transition) relation between the old state and the new state. Flexible (i.e., program) variables can either be primed or unprimed. In an action, unprimed occurrences of a variable are evaluated in the first state, whereas primed occurrences are evaluated in the second. For example, $x' + 1 = x$ is an action asserting that the value of x in the old state is one added to the value of x in the new state. The denotational semantics of an action A, written $[\![A]\!]$, is defined as a set of state pairs as expected.

TLA formulas consist of actions, logical operators, and temporal operators, and are evaluated over infinite traces. To give actions a meaning in the context of traces, we define their trace-semantics as follows:

$$[\![A]\!]^\omega \stackrel{\mathsf{def}}{=} \{s_1, s_2, s_3, \ldots \in \mathbf{State}^\omega \mid (s_1, s_2) \in [\![A]\!]\}$$

TLA includes the two temporal operators $\Box$ and $\Diamond$, with their usual meaning, where F is a formula:

$$[\![\Box F]\!]^\omega \stackrel{\mathsf{def}}{=} \{s_1, s_2, s_3, \ldots \in \mathbf{State}^\omega \mid \forall n \in \mathbb{N}.\, s_n, s_{n+1}, s_{n+2}, \ldots \in [\![F]\!]^\omega\}$$

$$[\![\Diamond F]\!]^\omega \stackrel{\mathsf{def}}{=} \{s_1, s_2, s_3, \ldots \in \mathbf{State}^\omega \mid \exists n \in \mathbb{N}.\, s_n, s_{n+1}, s_{n+2}, \ldots \in [\![F]\!]^\omega\}$$

Finally, we define (similarly to e.g. [25]) a *TLA contract* as a pair $C = (P, F)$, where P is a state predicate (the pre-condition), and F a TLA formula as above (the post-condition), with the following semantics:

$$[\![C]\!]^\omega \stackrel{\mathsf{def}}{=} \{\sigma = s_1, s_2, s_3, \ldots \in \mathbf{State}^\omega \mid s_1 \models P \rightarrow \sigma \in [\![F]\!]^\omega\}$$

5.2 Specifying Hoare-style Contracts with ACSL

For certain system components, one is not interested in their temporal behaviour, but rather in that they compute the correct outputs from the inputs (or formally, in the *state transformer* they embody). For this, it is sufficient to specify a relation between the pre- and post-states of the computation, and $\mathbf{State} \times \mathbf{State}$ is thus a suitable domain of runs. We will therefore use Hoare-style pre- and post-conditions to specify contracts for such components.

In ACSL, contracts are written as annotations in the source code, as C comments beginning with an @-symbol. Here, we shall limit ourselves to three ACSL constructs for writing contracts. The keyword `requires` specifies the pre-condition P of a function, which callers must satisfy before calling the function. The keyword `ensures` specifies the post-condition Q, which the function must guarantee upon return (w.r.t. partial correctness). The keyword `assigns` specifies the frame condition, i.e., a set of memory locations L that are allowed to change value during the call. In addition, ACSL supports so-called *ghost* variables, or logical variables, which may be used in specifications. Thus, viewing an ACSL contract as a triple $C = (P, Q, L)$, we define its semantics as:

$$[\![C]\!] \stackrel{\mathsf{def}}{=} \{(s, s') \mid \forall \mathcal{I}. (s \models_{\mathcal{I}} P \Rightarrow s' \models_{\mathcal{I}} Q) \wedge \forall l \notin L. s(l) = s'(l)\} \tag{3}$$

where $\mathcal{I}$ ranges over all possible interpretations of logical variables. We also note that any variables occurring in the pre-condition or frame condition, or parameters occurring in the post-condition, are evaluated in the pre-state. Variables (except parameters) occurring in the post-condition are evaluated in the post-state. The special keyword `\old` is used to enforce evaluation in the pre-state.

5.3 Abstraction to Denotational Contracts

We will now show how the concrete contracts, i.e., the TLA and ACSL contracts, can be abstracted into contracts in our abstract theory. The abstraction is based on the procedures' callees and the semantics defined in Sects. 5.1 and 5.2.

Definition 1 (From concrete to denotational contracts). *For procedure p calling other procedures P^-, equipped with concrete contract C_p, we define its* denotational contract $c_p \stackrel{\mathsf{def}}{=} (\rho^-_{c_p}, \rho^+_{c_p})$ *with interface $P^+_{c_p} \stackrel{\mathsf{def}}{=} \{p\}$ and $P^-_{c_p} \stackrel{\mathsf{def}}{=} P^-$, so that $\rho^+_{c_p}(p) \stackrel{\mathsf{def}}{=} [\![C_p]\!]$, and $\forall p' \in P^-.\ \rho^-_{c_p}(p') \stackrel{\mathsf{def}}{=} [\![C_{p'}]\!]$.*

In our instantiation of the contract theory, **Run** takes two different meanings, and thus the resulting denotations are over different semantic domains.

6 Implementing and Verifying Components

After having defined the components of the system, and specified them in the form of contracts, they can be implemented. The implementation is performed individually, under the assumption that the other components fulfil their contracts. This section presents a contract-relative semantics for the C language [16]. and shows how the example system can be implemented within it.

6.1 Infinite Traces over Program States

While we will not define here a full semantics for the C language (see e.g. [10] for a trace-based operational semantics for a subset of C, or [27] for a denotational-style semantics), this section describes how, given a semantics, the contract theory can be instantiated with C as the concrete implementation language, and specifically how to treat procedures as components. Programs are defined in the context of a finite set $\mathcal{P}$ of declared function names. We assume that function names are unique, i.e., there cannot be functions with the same name but with different types or number of parameters. The set of program states **State** is conventionally defined as the set of mappings from references to memory locations, and from memory locations to contents (as in, e.g., [10,26]).

For any finite set of function names P, we define the set of *function environments* $\mathbf{Env}_P = P \rightarrow 2^{\mathbf{Run}}$ containing the possible mappings from function names to traces, representing the effect of executing the respective function. *Interfaces* for sets of functions are defined as for components in Sect. 3. The semantics of a statement S is then defined in the context of an interface (P^-, P^+) and environments $\rho^- \in \mathbf{Env}_{P^-}$ and $\rho^+ \in \mathbf{Env}_{P^+}$, denoted $[\![S]\!]_{\rho^-}^{\rho^+}$, and $[\![S]\!]_{\rho^-}^{\rho^+} \subseteq \mathbf{Run}$. For example, the semantics of a function call without parameters, $[\![p();]\!]_{\rho^-}^{\rho^+}$, then depends on whether p is provided or required, and is defined as $\rho^+(p)$ if $p \in P^+$, and as $\rho^-(p)$ if $p \in P^-$.

Given an environment $\rho^- \in \mathbf{Env}_{P^-}$, we define the function $\xi : \mathbf{Env}_{P^+} \rightarrow \mathbf{Env}_{P^+}$ as $\xi(\rho^+)(p) \stackrel{\mathsf{def}}{=} [\![S_p]\!]_{\rho^-}^{\rho^+}$ for any $\rho^+ \in \mathbf{Env}_{P^+}$ and $p \in P^+$. Then, $(\mathbf{Env}_{P^+}, \sqsubseteq)$ is a complete lattice and ξ is monotonic, and thus, by Tarski's Fixed-Point Theorem [29], ξ has a least fixed-point ρ_0^+. In the context of an interface (P^-, P^+) and environment $\rho^- \in \mathbf{Env}_{P^-}$, we define $[\![S]\!]_{\rho^-} \stackrel{\mathsf{def}}{=} [\![S]\!]_{\rho^-}^{\rho_0^+}$ as the *standard denotation* of statements, where ρ_0^+ is the least fixed-point of ξ. Essentially, this ensures that the denotation of the function body of a procedure p is equal to the denotation of a call to p. In the above setting, we could instantiate **Run** to a variety of different domains (or combination thereof), such as **State** $\times$ **State** or $\mathbf{State}^\omega$.

6.2 Abstraction to Components

Using the above notation, we can now define an abstraction from C programs to components of our abstract contract theory.

Definition 2 (From C functions to components). *For any set of functions P^+ calling functions P', we define component $m : \mathbf{Env}_{P_m^-} \rightarrow \mathbf{Env}_{P_m^+}$ with $P_m^- \stackrel{\mathsf{def}}{=} P' \setminus P_m^+$ and $P_m^+ \stackrel{\mathsf{def}}{=} P^+$, by $\forall \rho_m^- \in \mathbf{Env}_{P_m^-}.\ \forall p \in P_m^+.\ m(\rho_m^-)(p) \stackrel{\mathsf{def}}{=} [\![S_p]\!]_{\rho_m^-}$.*

Following the scheme outlined in Sect. 4.1, we obtain a contract-relative semantics by using ρ_c as the relativising environment for the concrete semantics.

6.3 Procedure-Modular Verification of Components

When the components of the system have been implemented, the components can be verified against their contracts, assuming that the other components satisfy theirs. As long as the full contract-relative semantics of our concrete languages fulfils properties 2 and 3 stated in Sect. 4.1, and we have a proof system that is sound w.r.t. the semantics, the functions can be individually verified, and if this verification succeeds, the composition of all components is guaranteed to implement the top-level contract.

7 Designing the Example System

We now return to the example system introduced in Sect. 1, and design it according to the methodology proposed above.

7.1 Specifying the System

The task is to design a system fulfilling Property (1), and since we have shown how TLA can instantiate the contract theory, we use it to express the property:

$$F_{sys} = \Box\,((\texttt{in_temp}' = \texttt{sensor} \land \texttt{in_temp}' = t_0) \rightarrow\ \Diamond\,(\texttt{out_temp} = t_0 - 273))$$

Here, the variables `sensor`, `in_temp` and `out_temp` are (to be) program variables, or flexible variables in TLA terminology. They represent the sensor value, the read sensor value, and the value outputted to be displayed, respectively. The variable t_0 is a rigid variable, meaning that it is implicitly universally quantified, and used to relate read and displayed values in different states. The subformula $\texttt{in_temp}' = \texttt{sensor} \land \texttt{in_temp}' = t_0$ is a TLA action, denoting all state pairs where the value of `in_temp` in the second state equals the value of `sensor` in the first, with the intention of capturing that the sensor value has just been read. The subformula $(\texttt{out_temp} = t_0 - 273)$ is a state predicate, which is a special case of an action, only evaluated in the first state; we model displaying the temperature as writing it to the variable `out_temp`. Thus, the denotation $[\![F_{sys}]\!]^{\omega}$ will be the set of traces where for every value of `in_temp` read from the sensor, there eventually is a state where `out_temp` is assigned the converted value. In this case we will not impose any particular pre-condition, thus $P_{sys} = \texttt{true}$, forming the TLA contract $C_{sys} = (P_{sys}, F_{sys})$. In the abstracted denotational contract, we will have $P^{-}_{sys} = \varnothing$, since the full system should not depend on additional functions. For the provided functions, $\rho^{+}_{C_{sys}}(\texttt{main}) = [\![C_{sys}]\!]$, since `main` is the entry-point to the system, while for all other functions we allow any behaviour, since at the system level we are not interested in how they are implemented. Since displaying the temperature is only part of the system functionality, the full contract would be a conjunction of C_{sys} and several other contracts.

```
/*@ assigns in_temp;
    ensures in_temp ==
        (value >= 0 ?
            value : 0);
 */
```

(a) Contract C_{read} for `read`, reading and sanitising the sensor value

```
/*@ requires in_temp >= 0;
    assigns out_temp;
    ensures out_temp ==
        \old(in_temp) - 273;
 */
```

(b) Contract C_{conv} for `convert`, a temperature conversion function

Fig. 3. ACSL contracts for two functions in our system

7.2 Decomposing the System

As the next step, we want to decompose this system-level contract into contracts for the three functions of the system. The main function, which continuously calls the other functions, will be specified by the same contract, i.e. $C_{sys} = C_{main}$. However, in order for the main function to satisfy this, some assumptions (which are different from the precondition) on the helper functions are needed, essentially stating that the called procedures fulfil their respective guarantees. Thus, the two resulting denotational contracts will differ.

Two components then remain to be specified: a function `read`, reading and checking the sensor value, and a function `convert`, converting a temperature from Kelvin to Celsius. Contracts for these components, which we denote C_{read} and C_{conv}, are given in Fig. 3. In the case of `read`, we want the function to read a sensor value, given as a parameter `value`, and store it in `in_temp`, and the frame condition specifies therefore that this is the only variable assigned to. Furthermore, we want it to check that the value is valid, and if it is not lower than 0 (the lowest temperature in Kelvin) it should be written to `in_temp`, and otherwise be saturated to 0, before storing it. In the case of `convert`, we similarly specify its frame condition. As the post-condition we specify that it should store the converted value of `in_temp` in `out_temp`. However, it will not always be possible to compute this value, since there is the possibility of underflow for very small inputs. Thus, we also specify a pre-condition saying that the value must be non-negative, meaning that for negative inputs the result is left undefined.

The denotation $[\![C_{read}]\!]$ will then consist of all state pairs (s, s') such that when $s(\texttt{value}) \geq 0$ then $s'(\texttt{in_temp}) = s(\texttt{value})$ and otherwise $s'(\texttt{in_temp})$ will be 0, where in both cases no other variables are changed. The denotation $[\![C_{conv}]\!]$ will consist of all state pairs (s, s') such that $s(\texttt{in_temp}) \geq 0$ and $s'(\texttt{out_temp}) = s(\texttt{in_temp}) - 273$, or $s(\texttt{in_temp}) < 0$ and for all program variables $v \neq \texttt{out_temp}$, $s(v) = s'(v)$.

In the abstracted denotational contracts, we will have for the `main` function that $c_{main} = (\rho^-_{main}, \rho^+_{main})$, where $\rho^-_{main}(\texttt{read}) = [\![C_{read}]\!]$, $\rho^-_{main}(\texttt{convert}) = [\![C_{conv}]\!]$, and $\rho^+_{main}(\texttt{main}) = [\![C_{main}]\!]$. The functions `read` and `convert` will have no assumptions in their denotational contracts. Even without referring to a dedicated proof system, it should be obvious that $c_{main} \otimes c_{read} \otimes c_{conv} \preceq c_{sys}$; since both c_{sys} and c_{main} specify the same behaviour for `main`, then so will the

```
volatile int sensor;
int in_temp;
int out_temp;
void main() {
  while (1) {
    // ...
    read(sensor);
    // ...
    convert();
    // ...
  }
}
```

(a) Main program iteratively calling component functions

```
void read(int value) {
  if (value < 0) {
    in_temp = 0;
  } else {
    in_temp = value;
  }
}
```

(b) A function for reading and checking a sensor value

```
void convert() {
  out_temp = in_temp - 273;
}
```

(c) A function performing temperature unit conversion

Fig. 4. A simple C program consisting of three functions

composition, and since the composition restricts the other functions more than c_{sys}. In the composition, the dependencies on other functions will also be resolved, leaving no required functions. Thus, the composition is shown correct, and it remains to implement the components and verify them against their contracts.

7.3 Implementing the System

Figure 4 shows how the functions in the example system can be implemented in C. In a trace semantics setting, considering the three functions as a single program, we can see that calling `main` will produce strictly infinite traces, where `in_temp` is continuously assigned some value not less than 0, and shortly after that `out_temp` is assigned the converted value.

Let us now consider abstraction of components, individually, using different semantic domains as previously discussed. In the example, the `read` and `convert` functions do not make any procedure calls, meaning $P^-_{read} = P^-_{conv} = \varnothing$, thus $m_{read}(\rho^-)$ and $m_{conv}(\rho^-)$ are constant for all ρ^-. We then have $m_{read} \models c_{read}$ and $m_{conv} \models c_{conv}$, since their interfaces are compatible and, in the case of `read`, the denotations of the contract and the function body coindice exactly, i.e., $m_{read}(\texttt{read}) = c_{read}(\texttt{read})$, whereas for `convert`, since the contract leaves `out_temp` unspecified for $\texttt{in_temp} < 0$, its denotation is a superset of that of the function body, and $m_{conv}(\texttt{convert}) \subseteq c_{conv}(\texttt{convert})$. For the `main` function we consider a trace-based semantics, but where the mapping from the required functions `read` and `conv` has pairs of states as the domain. The scheduling function will then produce traces consisting of pairs of states, where the relation between the states within pairs is dependent on $\rho^-(\texttt{read})$ and $\rho^-(\texttt{convert})$, that is, for each function call is the assumed contract taken as a TLA action. Based on this, one can establish that $m_{main} \models c_{main}$ holds.

7.4 Verifying the System

In the example, the functions `convert` and `display_temp` are easily verified against their ACSL contracts using the existing Frama-C framework. This verification is in the context of state pairs, which can be considered the same domain as that of actions in TLA. Thus, considering the contracts for the above functions as TLA actions, the `main` function is verified procedure-modularly using TLC, by transforming such contracts into TLA actions, and, for the rest of the program, using standard procedures for converting programs into TLA specifications, as described, e.g., in [30]. Tools already exist that automate parts of this process, such as the Frama-C plugin C2TLA+ [22], which translates C programs to TLA. In our previous work [2], we extended this method to replace function bodies with their ACSL contracts at certain call sites, and, also with this method, the `main` function is easily verified against its contract. (There exist other tools for verifying C programs against temporal formulas [3,24], but we are not aware of any such tool that is procedure-modular.)

We have now shown that the composition of the sub-contracts refines the system contract, and verified that each component implements its contract. By the properties of the contract theory, then, we have established that $m_{main} \times m_{read} \times m_{conv} \models c_{sys}$, utilising existing verification techniques and tools, where components are individually implemented and verified.

8 Related Work

Modular design and formal reasoning about compositions of specifications and components have long been an active area of research. One early study is [1]. We mention here some additional works in the field. One main difference between these works and our work, is our treatment of procedures as the central unit of composition, enabling existing specification and verification frameworks to naturally be cast in our theory.

In [11], a compositional specification theory supporting assume-guarantee reasoning about the temporal ordering of input and output actions is presented. Components can be modelled either operationally, thus closely resembling the actual implementations, or more abstractly through declarative specifications. The theory includes a refinement preorder, enabling reuse of components, and operations such as parallel composition, conjunction, and quotienting, enabling independent and incremental development.

A generic model of contracts for embedded systems design is presented in [7], aimed at supporting a methodology of distributed development of different aspects of a system. The framework supports so-called “rich components”, in which a diverse set of functional and non-function aspects can be expressed and reasoned about. The contract-based formalisation of components makes an incremental realisation of components and viewpoints possible.

Inspired by the previous work, an axiomatisation of the notion of specifications is presented in [6], from which it is then shown how a contract framework can be built on such specification theories, by deriving the notion of contracts,

and their refinement and composition. The authors show that a trace-based contract theory can fit into this framework.

This is built upon in [12], extending the framework with synchronous and asynchronous (de)composition, with a focus on top-down design. To this end, the authors develop a proof system, in which the decomposition of assume-guarantee contracts generates proof obligations, which can be further reduced to satisfiability problems, and when verified one can conclude the correctness of the system. The framework can be instantiated with various temporal logics.

In [25], a trace-based logic for reasoning about the behaviour of While programs is presented. The framework includes a proof system that is both sound and complete. As a main result, it is shown that the logic subsumes the standard partial and total correctness Hoare logic, in the sense that in one direction, Hoare logic can be embedded in the presented logic, and in the other direction the trace-based logic can be projected onto standard Hoare logic, meaning that derivations in the former can be translated into derivations of the latter.

9 Conclusion

This paper presents a methodology for designing and reasoning about embedded procedural software. The formal basis is a contract theory we previously proposed in [21], generalised here by parametrisation on the semantic domain of runs, which is to be instantiated depending on the given properties of interest. The contract theory is applicable to real software in conjunction with existing formal verification tools, as illustrated by an instantiation with well-known, concrete specification and programming languages.

The proposed methodology follows the principles of *contract based design*, where high-level contracts are specified for the system as a whole, and decomposed into contracts for sub-components of the system, in different concrete languages and semantic domains. The implemented components are then verified *modularly*, thus, by virtue of the properties of the contract theory, ensuring that the high-level contracts are fulfilled.

Future work includes the extension of the theory with mechanisms for *interaction* with the environment, such as input/output, concurrency or message passing, in order for the theory to be useful in wider practical scenarios. This may require our trace semantics to be extended with an explicit notion of events, as well as with parallel composition. We also plan to instantiate the theory with a domain capturing the notion of time, such as timed words. Finally, we plan to develop a proof system for syntactically proving decompositions correct.

References

1. Abadi, M., Lamport, L.: Composing specifications. ACM Trans. Program. Lang. Syst. **15**(1), 73–132 (1993). https://doi.org/10.1145/151646.151649

2. Amilon, J., Lidström, C., Gurov, D.: Deductive verification based abstraction for software model checking. In: Margaria, T., Steffen, B. (eds.) Leveraging Applications of Formal Methods, Verification and Validation. Verification Principles. pp. 7–28. Springer International Publishing, Cham (2022). https://doi.org/10.1007/978-3-031-19849-6_2
3. Baranová, Z., et al.: Model checking of C and C++ with DIVINE 4. In: D'Souza, D., Narayan Kumar, K. (eds.) ATVA 2017. LNCS, vol. 10482, pp. 201–207. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-68167-2_14
4. Baudin, P., Filliâtre, J.C., Marché, C., Monate, B., Moy, Y., Prevosto, V.: ACSL: ANSI/ISO C Specification Language. http://frama-c.com/acsl.html
5. Baudin, P., Bobot, F., Correnson, L., Dargaye, Z., Blanchard, A.: WP Plug-in Manual - Frama-C 23.1 (Vanadium). CEA LIST. https://frama-c.com/download/frama-c-wp-manual.pdf
6. Bauer, S., et al.: Moving from specifications to contracts in component-based design. In: Fundamental Approaches to Software Engineering, pp. 43–58 (2012). https://doi.org/10.1007/978-3-642-28872-2_3
7. Benveniste, A., Caillaud, B., Ferrari, A., Mangeruca, L., Passerone, R., Sofronis, C.: Multiple viewpoint contract-based specification and design. In: Formal Methods for Components and Objects, vol. 5382, pp. 200–225 (Oct 2007). https://doi.org/10.1007/978-3-540-92188-2_9
8. Benveniste, A., et al.: Contracts for System Design, vol. 12. Now Publishers (2018). https://doi.org/10.1561/1000000053
9. Benvenuti, L., Ferrari, A., Mangeruca, L., Mazzi, E., Passerone, R., Sofronis, C.: A contract-based formalism for the specification of heterogeneous systems. In: 2008 Forum on Specification, Verification and Design Languages, pp. 142–147 (Sep 2008). https://doi.org/10.1109/FDL.2008.4641436
10. Blazy, S., Leroy, X.: Mechanized semantics for the Clight subset of the C language. J. Autom. Reason. 43 (2009). https://doi.org/10.1007/s10817-009-9148-3
11. Chen, T., Chilton, C., Jonsson, B., Kwiatkowska, M.: A compositional specification theory for component behaviours. In: Seidl, H. (ed.) ESOP 2012. LNCS, vol. 7211, pp. 148–168. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-28869-2_8
12. Cimatti, A., Tonetta, S.: Contracts-refinement proof system for component-based embedded systems. Sci. Comput. Program. 97 (2015). https://doi.org/10.1016/j.scico.2014.06.011
13. Cok, D.R.: JML and OpenJML for Java 16. In: Proceedings of the 23rd ACM International Workshop on Formal Techniques for Java-like Programs, pp. 65–67. Association for Computing Machinery, New York (2021). https://doi.org/10.1145/3464971.3468417
14. Floyd, R.W.: Assigning meanings to programs. Mathemat. Aspects Comput. Sci. **19**, 19–32 (1967). https://doi.org/10.1007/978-94-011-1793-7_4
15. Hoare, C.A.R.: An axiomatic basis for computer programming. Commun. ACM **12**(10), 576–580 (1969). https://doi.org/10.1145/363235.363259
16. ISO: ISO C standard 1999. Tech. rep. (1999). https://www.open-std.org/jtc1/sc22/wg14/www/docs/n1256.pdf, ISO/IEC 9899:1999 draft
17. Jones, C.: Specification and design of (parallel) programs. In: Proceedings Of IFIP 1983, vol. 83, pp. 321–332 (Jan 1983)
18. Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., Yakobowski, B.: Frama-C: A software analysis perspective. Formal Aspects Comput. **27**(3), 573–609 (2015). https://doi.org/10.1007/s00165-014-0326-7

19. Lamport, L.: The temporal logic of actions. ACM Trans. Program. Lang. Syst. **16**(3), 872–923 (1994). https://doi.org/10.1145/177492.177726
20. Lamport, L.: Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers. Addison-Wesley (June 2002)
21. Lidström, C., Gurov, D.: An abstract contract theory for programs with procedures. In: Guerra, E., Stoelinga, M. (eds.) Fundamental Approaches to Software Engineering, pp. 152–171. Springer International Publishing, Cham (2021). https://doi.org/10.1007/978-3-030-71500-7_8
22. Methni, A., Lemerre, M., Hedia, B., Haddad, S., Barkaoui, K.: Specifying and verifying concurrent C programs with TLA+. In: Formal Techniques for Safety-Critical Systems, vol. 476, pp. 206–222 (Nov 2014). https://doi.org/10.1007/978-3-319-17581-2_14
23. Meyer, B.: Applying "design by contract". IEEE Comput. **25**(10), 40–51 (1992). https://doi.org/10.1109/2.161279
24. Morse, J., Cordeiro, L., Nicole, D., Fischer, B.: Model checking LTL properties over ANSI-C programs with bounded traces. Softw. Syst. Model. **14**(1), 65–81 (2013). https://doi.org/10.1007/s10270-013-0366-0
25. Nakata, K., Uustalu, T.: A Hoare logic for the coinductive trace-based big-step semantics of While. Logical Meth. Comput. Sci. **11**(1) (2015). https://doi.org/10.2168/LMCS-11(1:1)2015
26. Nielson, H.R., Nielson, F.: Semantics with applications: an appetizer. Springer-Verlag, Berlin, Heidelberg (2007). https://doi.org/10.1007/978-1-84628-692-6
27. Papaspyrou, N.S.: Denotational semantics of ansi c. Comput. Stand. Interfaces **23**(3), 169–185 (2001). https://doi.org/10.1016/S0920-5489(01)00059-9
28. Staden, S.: On rely-guarantee reasoning. In: Hinze, R., Voigtländer, J. (eds.) MPC 2015. LNCS, vol. 9129, pp. 30–49. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-19797-5_2
29. Tarski, A.: A lattice-theoretical fixedpoint theorem and its applications. Pac. J. Math. **5**, 285–309 (1955)
30. Yu, Y., Manolios, P., Lamport, L.: Model checking TLA+ specifications. In: Pierre, L., Kropf, T. (eds.) Correct Hardware Design and Verification Methods, pp. 54–66. Springer, Berlin Heidelberg, Berlin, Heidelberg (1999). https://doi.org/10.1007/3-540-48153-2_60