

A Framework for Formal Specification of Embedded Systems*

Frantisek Bures, Ondrej Rysavy, Frantisek Scuglik, Miroslav Sveda
Department of Information Systems, Faculty of Information Technology
Brno University of Technology
Czech Republic
e-mail: {buresf, rysavy, scuglik, sveda}@fit.vutbr.cz

Abstract

This paper presents concepts of a specification-driven framework focusing on object-oriented design of embedded systems. It deals with tools and techniques aiming to prop formal specification and early design life cycle phases. Particularly, this contribution discusses logic calculus of objects, related class specification language, and techniques both for architecture specifications and for behavior specifications generated either from UML based semiformal behavioral diagrams or, as a reengineering tool, from source code.

1 Introduction

Embedded systems domain paradigm suggests considering both application requirements, namely time constraints defined by physical processes of the system environment, and implementation aspects, namely capacity constraints, from the beginning of system design. On the other hand, those constraints typically influence only small subset of the resulted solution while more important topics appear efficiency of design processes including techniques for reuse [10].

The contribution presents concepts of a specification-driven framework focusing on object-oriented design of embedded systems dealing with tools and techniques for formal specification and early design life cycle phases. Particularly, following sections of this paper discuss logic calculus of objects, related class specification language, and techniques both for architecture specifications and for behavior specifications generated either from UML based semiformal behavioral diagrams or, as a reengineering tool, from source code.

2 Specification Language

Specifying a system helps to understand it [3]. Specification is a written or graphical description (i) of what system is supposed to do (behavioral specs) or (ii) of system architecture (structural specs). A formal specification asserts that a description has precise and unambiguous semantics. The language of specification has to fit purposes of specification and be appropriate for a description of the system.

2.1 Logic calculus of objects

The formulation of the higher-order logic based on the type theory equipped only with object types is introduced in this section. It can be consider as the alternative of the simple theory of types defined by Church [12]. The language partially incorporates features of the object calculus with recursive types by Abadi and Cardelli [1]. Intention of these definitions is to prepare foundation for class specification language introduced in the next section. We do not provide exhaustive formal definition of the calculus, rather emphasis on the basic logical and operational features of the calculus will take place.

The inevitable part of calculus is a type system. It

* Supported by the Grant Agency of the Czech Republic through the grant GACR 102/05/0723: A Framework for Formal Specifications and Prototyping of Information System’s Network Applications
consists of the predefined type for propositions, denoted as $o$, and type constructor for simple and recursive object types. Type system contains all type terms constructible from the grammar given as following:

$$A, B ::= o \mid [l_i : A^{\text{iota}}_i] \mid \mu(a)A$$

The letters $A, B$ represent constructed types, $[l_i : A^{\text{iota}}_i]$ is a simple object type containing $n$ references attributed to distinct labels $l_i$ each of particular type $A$. Indexed letter $a_i$ stands for a type variable. It is exclusively involved in type terms as bounded variable for the recursive types, which is written as $\mu(a)A$ where type $A$ may contain free occurrences of the type variable $a$. An example given shows the simple object type containing three attributes of $\text{bool}$ type. It is more like a simple structure type than object.

$$[\text{red} : o, \text{yellow} : o, \text{green} : o]$$

More complex example represents object type that stands for a type of natural numbers. It involves the use of recursive object type for defining predecessor and successor numbers, or more precisely objects representing these numbers. Moreover the first attribute represents a flag which denotes an object encoding zero number:

$$\mu(a)\{\text{isZero : bool, pred : a, succ : a}\}.$$ 

The set of all terms of the language contains every finite sequence consists of the improper primitive symbols $ips = \{[,],=,\epsilon,\_\}$, variables $x_i$ from countable set of variable $V$, for every valid type $T$ and the collection $Q$ of equality symbols $\epsilon$ annotated by subscript for each type $T$. We distinguish the smallest subset of well-formed terms which contains terms that can be constructed using grammatical rules given as following:

$$m, n ::= x_i \mid \mu(a)\{x_i = \zeta(x : A)m^{\text{iota}}_i\} \mid m \downarrow \{x \leftarrow m\}$$

Operational semantic defines meaning for the calculus through imposing rules for computations. The schema of substitution using the form $M \[x \leftarrow N\]$, which represents a results of substitution where every free variable $x$ in $M$ is replaced with a term $N$, is exclusively used in these rules. Each object term is evaluated always to itself and represents a canonical term. The evaluation of attribute selection results to a term associated with the attribute in which all occurrences of bound variable is substituted for the whole object term $m \downarrow \{x \leftarrow m\}$ provided that.

$$m = [l_i = \zeta(x : A)m^{\text{iota}}_i] \text{ The evaluation of attribute update operation results to a new object in which all attribute except update one are the same} \text{ provided that}$$

$$m \downarrow \{l_i = \zeta(x : A)m^{\text{iota}}_i\} \text{ The last kind of terms represents equality relation. Two terms are equal if they can be reduced to the same canonical terms.}$$

The type system enables to distinguish another set of terms, called well-typed terms, which is subset of well-formed terms. These terms are valid according the type system if one can construct an evidence for a corresponding type judgement, which defines relation assigning type terms to language terms. Type systems represent logical environment for establishing these evidences by exploiting type rules enabling to infer all valid type judgements. All type rules of logic calculus of object is the same as the rules provided with Ob-cust that is detailed explained in\cite{11}.

The given calculus has raw and economical notation but expressible enough to represent constructions of functional higher-order logic calculus. It is done through definition of abbreviations. An abbreviation introduces a new name for particular construction either of type or language term. It enables us to use abbreviations in expressions instead of long and complex terms and their backward identification, which may simplify specification, computation and reasoning in the calculus. The list of three abbreviations introducing function type, function term and application of function is given as the example of the manner in which new constructions are incorporated into the language:

$$[A \rightarrow B] \quad \longrightarrow \quad [\text{arg} : A, \text{val} : B]$$

$$[\lambda(x : A), m] \quad \longrightarrow \quad [\text{arg} = \zeta(x : B)\text{.arg}, \text{val} = \zeta(s : B) [x \leftarrow s\cdot\text{arg}]]$$

$$[m(n)] \quad \longrightarrow \quad [m\cdot\text{arg} \leftarrow n, \text{val}]$$

The language is further extended to contain the truth constants $T, F$, logical connectives $\neg, \land, \lor, \Rightarrow, \Leftrightarrow$, and quantifiers $\forall, \exists$. In order to simplify writing terms the priorities for operators are specified such that negation has the highest priority and the existential quantifier has the lowest priority.

The calculus besides its computational meaning has also
defined the logical meaning, which has the form of deduction system. It consists of inference rules that enable one to construct proof for logical formulas that are theorems of the logic. Natural deduction system for classical higher-order logic can be, for instance, easily adopted to use with the calculus.

2.2 Class specification language

The logic calculus of objects provides a suitable formal environment for specifying and logical reasoning with properties of objects. However, writing specification directly in this calculus is tedious. The more appropriate notation is invented enabling to write neat specifications, but keeping possible to transform a specification easily to the calculus whenever required for reasoning.

**Classes.** A class is defined by specifying all of its visible properties. A property stands either for a field, which represents a state of an object, an observer, which serves for the read-only access to an object, or a modifier, which execution can change the state of an object. A field declaration includes the field name and the field class. Specification of a field may be concretized using invariant statement.

```
field fieldName : fieldClass
inv fieldInv = formula
```

The modifier method and the observer method include definitions consist of the name and arguments of the methods, and pairs of constraints. Constraints may involve variable self referencing to the actual object and any variable denoting the specified arguments of the method. Modifier method does not specify a result class, because it is implicitly considered to be a class in which modifier is defined.

```
Observer methodName(...,arg:argClass,...):retClass
  pre methodPre = formulaPre
  post methodPost = formulaPost
```

```
Modifier methodName(...,arg:argClass,...)
  pre methodPre = formulaPre
  post methodPost = formulaPost
```

**Inheritance.** The language enables define a new class by application of the single inheritance. An inherited class automatically receives all fields and methods of its parent class. To handle inheritance properly, a schema for definition of invariants and conditions of inherited fields and methods is needed. Considering that class B inherits a class A then the specification of classes has to preserve inheritance constraints assuming each inherited field and method in the schema as following:

\[
\text{fieldInv}^B \Rightarrow \text{fieldInv}^A
\]

\[
\forall\text{methodPre}^A \Rightarrow \text{methodPre}^B
\]

\[
\forall\text{methodPost}^A \Rightarrow \text{methodPost}^B
\]

Defining inheritance constraints in this manner enables method overriding. The precondition of the overridden method relaxes constraints of method execution, contrary to postconditions that involves additional constraints.

**Predefined classes.** Some common classes are defined directly in the logic calculus as they depend implicitly on propositional type and all possible instances are just represented by T and F objects declared previously as abbreviations. The class of natural numbers exploits recursive object type. It consists of three attributes, two of which serve as links to predecessor and successor objects. The iszero attribute marks the zero numeral.

```
boolType ≜ a
bool ≜ λ(b : boolType)b = T ∨ b = F
natType ≜ μ(ν)[pred : x, succ : x, iszero : a]
nat ≜ λ(n : natType)(n,iszero ⇒ n,pred = n)
  ∧ n,succ.p = n
```

The usual notation 0, 1, 2, . . . is explicitly used for denoting numerals. The possible representation of the numeral 0 is an object declared as following:

```
0 ≜ [iszero = T, pred = θ(n : natType)n,
    succ = θ(n : natType)(n,iszero ⇐ F),pred ⇐ n]
```

**Transformation to the formal logic.** Specification written in the class specification language is translated into the logic calculus following the next four rules.

- For each class to generate its object type. Each attribute has assigned object type, which corresponds to the declared attribute’s extent.

```
classType ≜ [field : fieldType,....,
            method : methodType,....]
```

- To translate field invariants to predicates. The contents of formulaInv remain unchanged. Note
that all free occurrence of variable self will become bounded.

\[ \text{fieldInv} \triangleq \lambda(\text{self} : \text{classType}) \text{formulInv} \]

- To translate preconditions and postconditions of methods into predicates and to compile them into the method predicate. The method predicate expresses the effect of the method execution as a relation between the states of the object, which precede the execution and stands for a result of the execution, respectively.

\[ \text{methodInv} \triangleq \lambda(\text{self} : \text{classType}) \\
\forall(\text{self} : \text{classType}) \forall(\text{arg} : \text{argType}) \\
(\text{self}.\text{method}(\text{arg}) = \text{self} \rightarrow \\
(\text{formulaPre} \Rightarrow \text{formulaPost}) \]

- To generate a class predicate in the form of logical conjunction of field predicates and method predicates. The definition restricts a set of instances of class to those which satisfy the predicate.

\[ \text{class} \triangleq \lambda(\text{self} : \text{classType}), \text{fieldInv}(\text{self}) \land ... \\
\land \text{methodInv}(\text{self}) \land ... \]

3 Formal Specification Support

The proposed development environment will cover concepts and tools of complete design and development life cycle of embedded systems aiming at industrial applications. Currently a toolset, including both freely available products and newly developed original tools, targets front-end parts of specification and design, namely formal specifications and rapid prototyping. The following subsections review original methods and tools developed to support not only proper specification of a new design, but also reuse of components developed in frame of previous applications of which formal specifications are not available.

3.1 Object-oriented architecture specifications

Section 2 provides a formal language as a tool intended for specifications with object-oriented features [5]. It directly represents formal foundations for basic object-oriented concepts. The language is expressive enough to provide more concepts of object orientation from the basic ones. Consequently, the specification language covers also language constructs for description of additional definitions and assumptions on specification in the form of logical formulas.

The specifications and assumptions are employed in the proof system that verifies whether a specification is valid under the given assumptions. To enable cooperation between the specification language and the proof system, we provide an abstract specification framework. The specification language is, from a structural point of view, composed from a language of predicate logic and the language of object calculus. By their synthesis we obtain a language with expressive power of higher-order logic. In terms of logic, the language consists of standard predicates logical symbols, i.e. quantifiers and propositional connectives, and constants, which are objects defined by terms of object calculus. Since terms are interpretable in the object calculus and the language allows quantification over the set of constants, the expressive power is equal to higher-order logic.

A specification consists of a set of classes that forms a model of the specified system. The reasoning about specification involves the use of some standard deduction system, such as Hilbert style deduction system as defined in section 2 or deduction system by Gentzen, which is more suitable for implementation in computer-based proof assistants. Because the specification language in this case is object-based, the classes are represented as special objects. A class is a basic structure of specifications that covers implementation of objects and logical judgments on properties of objects.

3.2 Behavioral specifications

We developed also tools for automated generation of CSP specifications [2]. The task is approached using either behavioral diagrams or application source code describing system behaviors. The behavioral diagrams stem from UML Composite States diagrams, where each diagram graphically describes a subsystem behavior. An n-ary tree is build for each diagram and, by climbing down the tree and recording the visited nodes, the tool generates the related CSP representation.
The automated translation from behavioral diagrams to CSP specifications is based on n-ary tree representation. This tree describes exactly the diagram structure and the mutual relations among particular diagram objects. Each node of the tree represents either an event or a process in CSP, so that when browsing the tree in correct order, the tool generates a CSP specification of the diagram as discussed below. Another dynamic structure, a dynamic list, stands for the synchronization points and communication channels.

The behavioral diagram consists of objects and mutual relations among the objects. Each object is in relation with one or more other objects. An n-ary tree provides the behavioral diagram representation in the tool’s run-time memory, in which root node of the tree represents process start, while the leaf nodes represent process ends. Nodes in between the root node and the leaf nodes stand for particular events of the behavioral diagram. Each tree has only one root node, but it may have more leaf nodes. A particular tree describes each process; e.g. when the system consists of three processes, then its representation contains three trees.

Each node is an object with particular information stored inside. Pointers to other nodes belong to this information. When performing depth-first search on the tree, the tool generates a CSP representation of the tree. Each node of each tree contains the name of an event or the name of a process. By climbing down all the trees and recording those names, the tool generates the related CSP representation of the complete system. Because the automated translation tool stores both the diagram and the tree, it is useful to include the diagram information directly in the tree. Concurrently, the positions of the graphical elements, i.e. process start, process end, and events, have to be stored also. Moreover, when an event is assigned to a mutual relation, it necessitates storing that event, too. The best way how to manage this information is to divide the tree into levels, where each even level represents process start, process ends or events, and each odd level represents mutual relations, see Figure 1.

Inclusion of the diagram structure directly in the tree increases amount of information stored, because each node contains also the graphical position of itself on the drawing area. Moreover, an object type, such as start, relation, event, should be included. The synchronization points are represented as dynamic lists. Each element of the list contains the synchronization alphabet and pointers to processes, which should synchronize. A dynamic list represents also the communication channel but each element contains only the type of the channel, without pointers. The indication that a process communicates among the channel is stored directly in the n-ary tree’s proper node.

3.3 Reverse specifications

The translation from application source code into CSP specification stems from grammar-based compiler techniques [6]. The grammar exactly describes which source code structures the compiler accepts. Applying corresponding syntax analysis, the compiler transforms input source code into compiler’s inner variables. Using those variables together with the knowledge of their meaning, the compiler generates the related CSP specification.

Each compiler grammar exactly defines, which source code structures the compiler accepts. The demonstration grammar recognizes only basic C program statements such as function call, conditional, and cycle. Recursive statements are accepted also. To simplify the grammar, some details of syntax are omitted. Standard rewriting rules define the grammar, where symbols written with lowercase letters represent terminals and symbols written with uppercase represents nonterminals.
The grammar follows:

\[ S \rightarrow \text{IF | WHILE | id IDCONT} \]
\[ \text{IF} \rightarrow \text{if (condition) CODE IFCONT} \]
\[ \text{IFCONT} \rightarrow \text{else CODE | e} \]
\[ \text{CODE} \rightarrow S | \{\text{BODY}\} | ; \]
\[ \text{IDCONT} \rightarrow \text{= value; | (params);} \]
\[ \text{BODY} \rightarrow S \text{ BODY | e} \]
\[ \text{WHILE} \rightarrow \text{while (condition) CODE} \]

The compiler browses the input source code on the fly respecting the grammar. When top-down parsing reaches a terminal symbol, the compiler generates the related CSP specification of previously processed source code. The compiler performs syntax analysis of the source code on its input. Whenever the compiler reaches a terminal symbol in the grammar, the CSP specification of this terminal is generated. The symbols processed before are stored in compiler’s inner variables to support generating the CSP specification. Parsing the source code, the compiler pushes each occurrence of a symbol on its stack. When generating the related CSP code, the symbols are popped from the stack.

The following paragraphs describe how to transform the grammar’s terminal symbols into CSP representation (each terminal symbol is followed by its CSP representation). Note that PROC1 stands for the process name popped from the stack, PROC2 stands for the name of the process following the current process. This name is pushed on the stack. If no more program statements occur in the input source code, then PROC2 = SKIP

- value;
  PROC1 = assign_value → PROC2

{params};
  PROC1 = process_params → MY FUNC
  MY FUNC = end_of_function → PROC2
  - event end_of_function represents the end of a function stored by the id terminal symbol processing

) or ;
  PROC1 = cond hold → PROC3
  - PROC3 stands for the name of the process representing the BODY; it is popped from the stack.

) else or ; else
  PROC1 = cond hold → PROC3
  | else → PROC2
  - PROC3 stands for the name of the process representing the BODY; it is popped from the stack.
  PROC2 = the name of the process representing the BODY of the else part; it is pushed on the stack.

Other terminal symbols used in the grammar do not generate CSP specification directly. These symbols initiate only pushing process names on the stack and storing related variables into the symbol table.

3.4 Shared Variables

A tiny extension of this minimal core of CSP deals with communication among processes. The CSP algebra includes state variables locked in one process while not accessible from other processes. Inter-process communication through shared variable can be emulated so that two processes VAR and VAR2 communicate among the same channels and have the same labels:

\[ VAR = \text{left?value → VAR(value)} \]
\[ | \text{right!value → right2!value → VAR} \]
\[ VAR2 = \text{right2?value → left!value → VAR2} \]

More detailed specification on shared variables in CSP is presented in [13].

4 Case Study

This section demonstrates utilizations of the discussed specification approach by a real-world case study. After succinct and informal specification of the overall system’s environment, the succeeding subsections deal with a formal treatment of structural and behavioral specifications.

4.1 Case study intro

Cruise Control (CC) relieves the driver of the responsibility to maintain a constant speed if greater than 40 kmph. The CC system also enables measurement of the actual speed and lifetime kilometers of the car. The driver can activate and deactivate CC by pressing the appropriate button. When the driver gives an activate command, the actual speed of the car is maintained. The driver may deactivate the CC system at any time by a deactivate command. When the driver increases/decreases the speed of the car by pressing the gas/brake pedal, the cruise control system goes off. When the pedal is released, the system regains control
and maintains the new actual speed.

The Cruise Control receives pulses from the rotation sensor and compares the frequency of those pulses to the frequency value stored in its memory, and it sends pulses to a throttle actuator. The system counts pulses from its own sensor on the drive shaft. Count-rate corresponds to vehicle kilometers per hour through proportionally. The system displays this value on the speedometer. The speedometer includes a counter that counts and displays the lifetime kilometers driven by car. When the car’s speed increases (decreases) the frequency of the pulses increases (decreases). For any given speed of the car there is a corresponding pulse frequency.

The throttle actuator is adjusting the throttle position. The CC actuates the throttle valve by a cable connected to an actuator, instead of by pressing the gas pedal. The brake sensor indicates when the brake pedal is press. The accelerator sensor indicates when the gas pedal is press or release. The panel shows actual speed (speedometer), lifetime kilometers of the car (odometer), and cruise indicator light. The odometer counts 1-kilometre units and displays the value. The cruise indicator indicates if cruise control is active.

4.2 Decomposition specification

In this section we use the class specification language for decomposition of the cruise control system from our running example. Decomposition simplifies the specification by separating it into classes which can only interact through defined interfaces. Constraints associated with attributes define valid instances of a class but does not explicitly restrict behavior of objects. The specification of the Panel class stands for the speed displaying unit of the system. The class is simplified to have only the speed attribute and a method that updates the speed value.

```plaintext
class Panel {
begin
  field speed : nat
  modifier UpdateSpeed(arg:nat)
    pre UpdateSpeedPre = arg ≤ maxSpeed
    post UpdateSpeedPost = self'.speed = arg
end
```

The core class of the specified system is the cruise control class. An instance of this class represents a single cruise control unit. For the sake of simplicity the cruise control class contains only the status flag, the two fields storing actual speed and maintained speed, the associated throttle and panel objects, and the three methods for activation, deactivation of the control and maintain selected speed.

```plaintext
class CC {
begin
  field panel : Panel
  field throttle : Throttle
  field isActive : bool
  field actSpeed : nat
  field selSpeed : nat
  modifier Activate
    pre activatePre = self.actSpeed ≥ 40
    and not self.isActive
    post activatePost = self'.actSpeed = self'.selSpeed
      and self'.isActive
  modifier Deactivate
    pre deactivatePre = self.isActive
    post deactivatePost = not self.isActive
end
```

The throttle class specifies the throttle actuator unit. Again, the radical simplification led to trivial class definition, in which only the throttle level field remains.

```plaintext
class Throttle {
begin
  field level : nat
  inv levelInv = 0 ≤ self.level and
    self.level ≤ 100
  modifier SetLevel(arg:nat)
    post level = arg
end
```

All classes in the following group are the sensor classes. They are inherited from the abstract sensor class which contains only the field cc that refers to a control object. The RotationSensor class specifies the unit of the same name that does measurement of the driving speed. The measured speed is stored and publicly available in the system. A pedal sensor indicates the state of a pedal in its isPressed field.

```plaintext
class Sensor {
begin
  field cc : CC
end
class RotationSensor inherits Sensor {
begin
  field speed:nat
  modifier Measure : RotationSensor
end
```

```plaintext
class PedalSensor inherits Sensor {
begin
  field isPressed : bool
end
```

```plaintext
class BrakePedal inherits PedalSensor
```

```plaintext
class AccelPedal inherits PedalSensor
```

The decomposition identifies the system as a collection of classes. It uses constraints on attributes in the form of
predicates to declare static properties of classes. The specification can be further refined by extending of classes with other attributes and specifying additional constraints or utilized to declare behavioral specification for individual classes.

4.3 TLA specifications

Behavioral specification of the Maintain method is written in TLA. The CC class is considered as a set of objects, thus, the usual notation can be used for declaring a variable cc. The constant div represents a maximal divergence between actual and maintained speed. Declaration of type invariant is required because TLA is the untyped language.

```
module Maintain
begin
variable cc
constant div
TypeInvariant defas cc ∈ CC
Reduce defas (cc.actSpeed - cc.selSpeed) > div ∧ cc'.throttle.level < c.throttle.level
Accelerate defas (cc.actSpeed - cc.selSpeed) < -div ∧ cc'.throttle.level > cc.throttle.level
Maintain defas Init ∧ (Reduce ∨ Accelerate)
theorem Maintain ⇒ □TypeInvariant
end
```

4.4 Behavior specifications

Behavioral diagrams can describe required behaviors of the CC class. The class includes five public variables (only four of them can influence the behavior of the class) and four methods described by four processes in the behavior specification. Each process has to grasp the pre and post conditions denoted in the class specification. Figure 2 depicts the behavioral diagram of the CC class.

The method mentioned in Subsection 3.2 can generate from this behavioral diagram the following CSP specification.

```
isActive : VAR
selSpeed : VAR
actSpeed : VAR
speed_error : VAR
ACTIVATE = activate ⇒ isActive.lock ⇒
          isActive.left!1 ⇒ isActive.unlock ⇒
          actSpeed.right?x ⇒ selSpeed.lock ⇒
          selSpeed.left!x ⇒ selSpeed.unlock ⇒
          MAINTAIN_SPEED
MAINTAIN = isActive.right?x ⇒ compare.0.x ⇒
```

Figure 2. Behavioral diagram of the CC class
equal → ACTIVATE
| less → MAINTAIN_SPEED )
| deactivate → ACTIVATE
| suspend → resume → MAINTAIN
MAINTAIN_SPEED = speed.right?act →
selSpeed.right?main → speed_error.right?err →
compare.act.main
→ ( equal → MAINTAIN
| greater → compare.(act-main).err
→ ( less → MAINTAIN
| equal → MAINTAIN
| greater → level.right?fuel →
throttle.setLevel.fuel-1 →
MAINTAIN )
| less → compare.(main-act).err
→ ( less → MAINTAIN
| equal → MAINTAIN
| greater → level.right?fuel →
throttle.setLevel.fuel+1 →
MAINTAIN )
)
DEACTIVATE = deactivate → isActive.lock →
isActive.left!0 → isActive.unlock →
DEACTIVATE

The second diagram depicted in Figure 3 describes the behavioral specification of the Throttle class. The followed CSP specification is generated using the method from Subsection 3.2, too.

![Figure 3. Behavioral diagram of the Throttle class](image)

4.5 Reverse specifications

The presented reuse approach utilizes a technique for automated generation of a formal specification for the source code that enables to verify reused parts of the system. As an example in frame of this case study, we demonstrate the reuse of the SetLevel method from the Throttle class. Let’s suppose that the source code of the reused SetLevel method is defined as follows:

```java
function SetLevel (int fuel) {
    global int level;
    level = fuel;
}
```

Then, the generated specification in CSP of the denoted source code is following:

```
4 B Level : VAR
5 B SETLEVEL(fuel)= throttle.setLevel.fuel→
level.lock→
level.left!fuel→
level.unlock→
SKIP
```

The difference between the CSP specification obtained from the behavioral specification and using the reverse specification is caused by the isolated function in the reversal specification on the contrary to the cooperating specifications of the CC class and the Throttle class.

5 Conclusions

Perri and Kaiser [4] formulate a model of development environments employing three tool types: (i) structures as reusable components embodied into developed systems, (ii) mechanisms as proper development tools used for development process (but not included into developed systems), and (iii) strategies as design and development methods. Solving problems improves acquired skills by enhancing experience that has the property of being reusable. Every design deserves decisions based on an application domain knowledge, which includes facts about previous similar implementations. This domain knowledge can be delivered to the design process by tool types in the following way: (i) by reusable component libraries [9], (ii) by dedicated design and development tools [8], and (iii) by stepwise development of the original practice saving the successful cases in a knowledge-based subsystem, see [7].

The presented framework aims at complementing the above mentioned features of development environments by additional support for the utilization and reuse of formal specifications in embedded systems design. Particularly, it employs logic calculus of objects, class specification language, and techniques both for architecture
specifications and for behavior specifications generated either from UML based semiformal behavioral diagrams or, as a reengineering tool, from source code.

References


