# The Quest for Correct Systems: Model Checking of Diagrams and Datatypes [*]

Jan Philipps and Oscar Slotosch

Institut für Informatik
Technische Universität München
80290 München, Germany
{philipps,slotosch}@in.tum.de

## Abstract

*For the practical development of provably correct software for embedded systems the close integration of CASE tools and verification tools is required. This paper describes the combination of the CASE tool* AutoFocus *with the model checker SMV.* AutoFocus *provides graphical description techniques for system structure and behavior. In* AutoFocus*, data types are specified in a functional style, while SMV supports only primitive data types. Hence, a data type translation based on the techniques used in compiling functional programming languages is a major part in the mapping from* AutoFocus *to SMV.*

## 1. Introduction

Common techniques for the quality assurance of software are code reviews and tests. Both techniques, however, fail to ensure the high levels of quality required for software in embedded systems, where human life or property may be at stake, or malfunctions can lead to expensive product recalls.

Formal methods, on the other hand, allow to *prove* the correctness of a program based on mathematical models of both the program and the correctness criteria. The mathematical formalization of programs and properties itself is a possible source of errors, therefore it is done by experts in formal methods in order to ensure the adequacy of the model and the properties.

There are various approaches to close this gap. One is to base formal methods on common mathematical principles —e.g. set theory— that are either known to the system designers or that are easier to master. Examples for this approach are the specification methods B [1] and Z [23]. In both cases, tool support is secondary, and correctness proofs must be done manually. Neither is there support for graphical notations for system structure or behavior.

Another approach is to avoid mathematical notation by defining formal semantics for the programming and modeling languages typically used in industry. Examples are formalizations of UML [4], SDL [10] or StateCharts [8]. Unfortunately, even when the formalization is sufficiently precise so that verification support can be provided, the formalization is often too complicated for practical, perhaps even automatic, verification.

A third approach is to use domain-specific programming languages and to provide verification support for them. Examples are the model checker SPIN [11] with its programming language PROMELA for communication protocols or model checkers for subsets of the hardware description languages Verilog and VHDL, as for example VIS [24] or the Cadence version of SMV. These systems, however, have no support for more intuitive graphical description techniques; they also have no support for data types more complex than (tuples of) booleans, finite subsets of the integers and enumerations.

This paper describes an attempt to address the deficits of the approaches mentioned by closely integrating graphical notations, straightforward semantics and complex data types, based on the CASE tool AutoFocus [12, 13]. AutoFocus offers graphical, hierarchical and view oriented descriptions of systems. In contrast to other approaches the semantics of AutoFocus is based on elementary mathematical concepts, which allows us to use existing automatic verification tools, in this case the model checker SMV. Like most automatic verification tools SMV is restricted to rather simple data types, while AutoFocus allows powerful data specifications in the style of functional programming languages. Therefore, data types must be mapped to simple

[*] This work is supported by the Bundesamt für Sicherheit im Informationswesen (BSI) within the project Quest and by the DFG within the Sonderforschungsbereich 342.

integer subsets and finite enumerations for verification.

The paper is structured as follows. In the next section, we introduce the AutoFocus description techniques with a small avionics example. § 3 contains a short overview of the SMV features that are used for the translation from AutoFocus to SMV. The main part of the paper are the translation of the user-defined data types and functions in § 4 and the translation of the graphical description techniques for system structure and behavior in § 5. § 6 contains some results of a larger example; in the conclusion (§ 7) we mention extensions of our approach and give an outlook on further work.

## 2. AutoFocus

AutoFocus [12, 13] is a CASE tool for the development of correct embedded systems software. In embedded systems a software part runs embedded into a electric or mechanical hardware environment. They often follow a cyclic operation model, similar to the clocked hardware model. Concepts from object-orientation are not as dominant as in other areas of software design.

In AutoFocus systems are specified as hierarchical dataflow graphs; the components themselves are basically extended Mealy machines. This helps to keep the semantics of AutoFocus clean and simple. The data types of component states and communication channels are specified in a style similar to that of functional programming languages [3, 9, 19].

In addition to the editors for system specification, AutoFocus has a *consistency wizard* [14] which allows the designer to define and check static consistency conditions of system descriptions. A simulation environment can be used for rapid prototyping.

For a detailed description of AutoFocus we refer to [12, 13]; in this paper we explain the relevant views of AutoFocus using a simple avionics example.

### 2.1. Example

Our example is an alarm management system for a two-engine aircraft; it is a simplification of a case study from the project "KorSys" (*correct systems*, [7]). The system collects alarms from the engines and displays them to the pilot together with a list of instructions how the pilot should deal with the alarm; for example, the pilot might shut down an overheated engine.

The pilot can select two display modes using function keys **NAV** and **AC**. In *navigation mode*, the pilot can switch between different views related to the position of the aircraft on a map. In *alarm control mode*, the pilot deals with the alarms. In each mode different function keys are enabled. In the navigation mode the pilot can switch between information displays with function keys **F1**, **F2**, and **F3**. In the alarm mode the pilot can browse the "do-lists" (holding the instructions for possible corrective actions) using a **Do** key and acknowledge actions and alarms using an **Ack** key.

If the alarm management system receives an alarm, the display automatically switches into the alarm control mode. The main task of the avionic alarm control is to ensure that the pilot receives all incoming alarms, that the latest alarm is displayed in front of other pending alarms and that alarms are stored until they have been explicitly acknowledged by the pilot.

In the rest of this section, we demonstrate the AutoFocus description techniques for system structure, system behavior and data types.

### 2.2. System Structure

The system structure is described by system structure diagrams (SSDs), which describe the component interconnection, the syntactical interfaces of the components and the interface data types. Figure 1 shows the structure of the avionics system. It has two subcomponents, one to control the display, and the other to store the alarms. AutoFocus descriptions can be hierarchical, and the components could be refined to further SSDs.

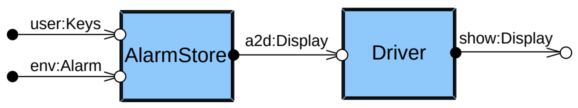

**Figure 1. System Structure Diagram**

The external channels in Figure 1 describe the interface of the alarm system. The interface consists of three channels:

- `user:Keys` receives key signals (**F1**, **F2**, **F3**, **Do**, **Ack**, **NAV** and **AC**) from the pilot to change the displayed information and to control the reactions on the occurrence of alarms.
- `env:Alarm` receives the alarm messages from the environment. There are three kinds of alarms: "Fuel", denoting that an engine has no fuel, "Temp", if an engine is overheated, and "PonR", signaling that the point of no return has been reached, i.e. that the fuel does not suffice to return home.
- `show:Display` is the output to the display. The display presents general flight information, the engine state (all engines operating, one engine inoperative, all engines inoperative) and the list of alarms, together with instructions to handle the most recent alarm.

```
// description of alarm type and alarm store
      data Engine = Eng1 | Eng2;
      data Alarm = None | Fuel(Engine) | Temp(Engine) | PonR;
      data AlarmStore = Store(Bool,Bool,Bool,Bool,Bool);
// constant for an empty alarm store:
      const EmptyStore = Store(False, False, False, False, False);
// description of keys
      data FunctKeys = F1 | F2 | F3 | Do | Ack;
      data ContrlKeys = AC | NAV;
      data Keys = FK(getFK:FunctKeys) | CK(getCK:ContrlKeys);
// description of engine state
      data EngineState = AllEngOp | OneEngInop(Engine) | AllEngInop;
// abstract description of display information
      data Actions = Refill(Engine) | SwitchOff(Engine) | FindOtherBase | GoDown;
      data NavigationModes = Sector | Rose | North;
      data Display = Nav(NavigationModes,EngineState)
                   | Err(AlarmStore,EngineState)
                   | DoList(Actions,EngineState);
```

**Figure 2. Data type declarations**

## 2.3. Data Types

The types of the interfaces are defined using a functional specification style, similar to the functional programming languages Haskell or ML [9, 19]. They have the following general form:

$$\begin{aligned} dt = \; & C_1(sel_{1,1} : typ_{1,1}, \dots, sel_{1,k} : typ_{1,k}) \mid \dots \mid \\ & C_n(sel_{n,1} : typ_{n,1}, \dots, sel_{n,m} : typ_{n,m}) \end{aligned}$$

The $C_i$ are constructor functions, the $sel_{i_j}$ are selector functions. The constructor and selector names have to be different. If selectors are omitted in the definition, default names are generated. Moreover, for every constructor $C_i$ an implicit discriminator function $is\_C_i$ is generated.

In our avionics example, the alarm type and the alarm store type are defined as shown in Figure 2. Only for the data type `Keys` are explicit selector functions defined. The figure also contains the definition of a constant `EmptyStore`.

Functions or predicates over data types are also specified as in functional programming languages. In particular, they can be defined using pattern matching on the left hand side of a definition. For example, the avionics system uses a function `insertAlarm`, defined by:

```
fun insertAlarm(Fuel(Eng1),Store(e1,e2,t1,t2,p))
        = Store(True,e2,t1,t2,p)
  | insertAlarm(Fuel(Eng2),Store(e1,e2,t1,t2,p))
        = Store(e1,True,t1,t2,p)
  | insertAlarm(Temp(Eng1),Store(e1,e2,t1,t2,p))
        = Store(e1,e2,True,t2,p)
  | insertAlarm(Temp(Eng2),Store(e1,e2,t1,t2,p))
        = Store(e1,e2,t1,True,p)
  | insertAlarm(PonR,Store(e1,e2,t1,t2,p))
        = Store(e1,e2,t1,t2,True);
```

Components can also have local data state variables. The local variables of the component AlarmStore are listed in Figure 3.

| Name | Type | Initial value |
|---|---|---|
| `as` | `AlarmStore` | `EmptyStore` |
| `es` | `EngineState` | `AllEngOp` |
| `last` | `Alarm` | `None` |

**Figure 3. AlarmStore variables**

## 2.4. Behavioral View

In AutoFocus—as in many other CASE tools— the behavior of components is defined by state machines; the corresponding diagrams are called state transition diagrams (STDs).

We skip the STD for the driver component, as its function is quite simple: It just ensures continuous output of display information to the hardware, even if no new signals arrive from the alarm store. The STD for AlarmStore is shown in Figure 4. It has two states, each corresponding to a display mode. Each transition label consists of four parts, separated by a colon (":"): a *precondition*, one or more *input statements*, one or more *output statements* and an *action*. Each part is optional.

The precondition can restrict the execution of a transition depending on the value of the data state variables; the action changes these variables. Variable names that are used in the transition label but that are not component variables, are called *transition variables*.

Similar to function definitions, transitions in AutoFocus use pattern matching to read from the input channels and to bind transition variables. Preconditions, output statements, and actions can be expressed using user-defined and automatically generated functions from the data type specification.

For example, in Figure 4 one of the transitions from NAV into AC uses the function `insertAlarm` defined in § 2.3. The input statements of a transition, too, can use pattern matching: We can use the more compact `user?CK(NAV)` instead of adding `is_CK(x) & getCK(x) = NAV` to the precondition together with the input statement `user?x`.

Like structure diagrams, STDs can be hierarchical. In our example, the description of the two modes is refined using sub-STDs automata that describe the behavior in each state in more detail. Figure 5 shows the sub-STD of the alarm control state AC.

For each incoming and outgoing transition in the enclosing state transition diagram (cf. Figure 4) there is a connector, represented as a small bullet. A transition from or to a connector is an entry or exit transition of the sub-STD. Multiple transitions may lead to or from an exit (empty bulllet) or entry (black bullet) connector. In the example the exit connector is connected to three internal transitions, since the alarm control mode can be left from all connected states if the key **NAV** is pressed.

Figure 5 also shows that transitions can use abbreviated labels, to avoid cluttering the STDs with the full label definition. StoreFirstAlarm, for example, is the abbreviated label for the following transition:

```
isEmptyStore(as) :
env?al :
out!Err(insertAlarm(al,as),es) :
as = insertAlarm(al,as)
```

where `isEmptyStore` is a user-defined function that checks for equality with the constant `EmptyStore` from Figure 2.

## 3. The Model Checker SMV

A model checker is a program that verifies whether a given system model (described in a simple programming language) satisfies a property specification (described in a temporal logic). This verification is done not deductively, but rather by exhaustively examining every possible behavior of the system. If the property does not hold for a system, the model checker produces a counter example, i.e. a trace of a system execution that violates the property.

We choose SMV [17] as the verification tool, mainly because its semantics is close to the synchronous execution model of the current AutoFocus implementation; other verification systems such as SPIN or STeP aim more at interleaving execution models. SMV is a symbolic model checker, i.e. it uses BDDs for the representation of state sets, and does not store states explicitly in a hash table. This is claimed to be more efficient for hardware-like systems. Last but not least, SMV is freely available and easily portable to other platforms.

The typical application field of SMV is hardware verification. This is reflected in its input language, which allows concise specifications of combinatorial hardware and registers, but offers no direct support for more complex data types, function declarations, or state machines. In § 4 and § 5 we show how these higher-level concepts can be mapped to SMV.

In this section, we give an overview over those parts of the model and property description languages of SMV that are relevant to this paper; more information about SMV as well as the system itself is available from the SMV web site: `http://www.cs.cmu.edu/~modelcheck/`.

### 3.1. Model Language

Fundamentally, an SMV model is a specification of a finite state machine. The state space of this machine consists of variables that hold either a Boolean value, a member of an enumeration, or an element of a finite integer range. For example,

```
VAR b: boolean; VAR e: { e1, e2 }; VAR i: -5..5;
```

The initial state of the model is specified by a formula in propositional logic that refers to the values of the state variables. For Boolean variables the usual connectives can be used; for integer values comparisons and arithmetic are available.

The transition relation is also specified by a formula that in addition refers to the values of the state variables in the following state. For example, given the declarations above, the formula

```
next(b) = !b & next(e) = e1 & (i = 0 | i = 1)
```

specifies a transition relation from all states with $\mathtt{i} \in \{0, 1\}$ to states where $e$ equals $e1$, $\mathtt{b}$ is inverted, and the value of $\mathtt{i}$ is arbitrary.

Obviously, this relation is not total: there are no transitions from states where $\mathtt{i}$ is less than zero, or larger than one. It is the task of the model designer —or, in our case, the compiler from AutoFocus to SMV— to ensure that transition relations are consistent and total.

To structure larger formulas, SMV allows `case`-expressions:

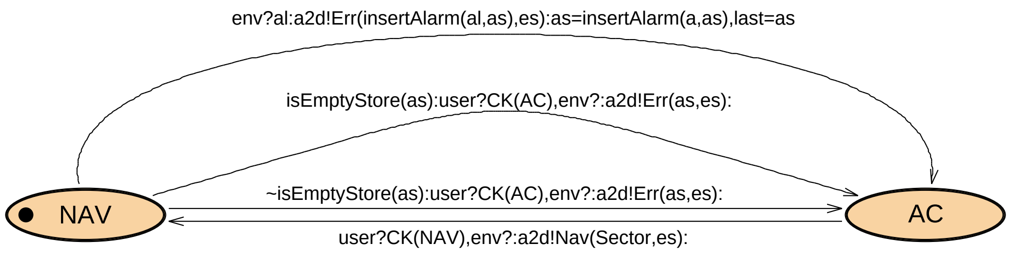
env?al:a2d!Err(insertAlarm(al,as),es):as=insertAlarm(a,as),last=as
isEmptyStore(as):user?CK(AC),env?:a2d!Err(as,es):
~isEmptyStore(as):user?CK(AC),env?:a2d!Err(as,es):
user?CK(NAV),env?:a2d!Nav(Sector,es):
NAV
AC

**Figure 4. Behavior of AlarmStore**

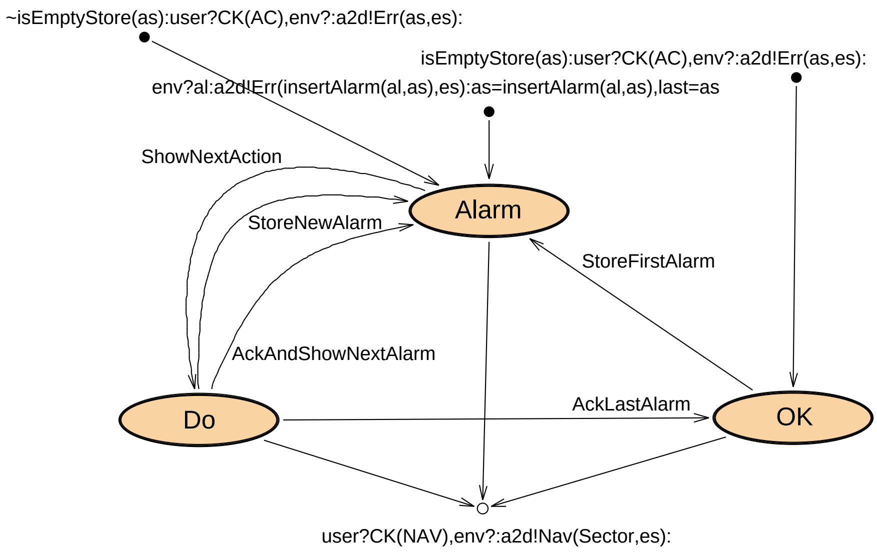
~isEmptyStore(as):user?CK(AC),env?:a2d!Err(as,es):
isEmptyStore(as):user?CK(AC),env?:a2d!Err(as,es):
env?al:a2d!Err(insertAlarm(al,as),es):as=insertAlarm(al,as),last=as
ShowNextAction
StoreNewAlarm
Alarm
StoreFirstAlarm
AckAndShowNextAlarm
AckLastAlarm
Do
OK
user?CK(NAV),env?:a2d!Nav(Sector,es):

**Figure 5. Behavior of AC**

```
  case
    c1 : e1;
    c2 : e2;
    ...
    cn : en;
  esac;
```

is an abbreviation for:

```
  (c1 -> e1) & (!c1 & c2 -> e2) &
      ...    & (!c1 & !... & cn -> en)
```

Note that since the cases are prioritized, the value of the `case` is the first $e_i$ with $c_i$. Frequently, a default case is introduced, by choosing “true” (in SMV: `1`) for $c_n$.

For the specification of larger systems, SMV models can be divided into several modules. Modules can be parameterized, and each module can have local variable declarations, its own initialization and its own transition relation. The modules are instantiated in a main module similar to state variable declarations.

Semantically, the modules operate in parallel: the initialization predicates of all modules and their transition relations are conjoined.

## 3.2. Property Language

The second input language of SMV is the property specification language. Properties are formulated in the *computation tree logic* CTL. A detailed description of CTL formulas and their semantics can be found in [6].

In general, it is hard to formulate system properties in temporal logics; [16] contains a classification of typical property specification patterns and their formalization in CTL and other languages.

## 4. Data Type Translation

In this section we describe the translation from user-defined types and functions into SMV. The concept of the translation is to define a mapping from all values of the user-defined type to integer numbers, and to use case statements instead of function calls. Of course this concept works only for finite types and functions, but model checking generally is restricted to finite systems.

The translation is based on the following two steps. First, functions are normalized by generating symbolic constants for all values of each data type used in the AutoFocus model, for all standard functions (constructors, selectors, discriminators) and for all user-defined functions.

Then, the symbolic constants are eliminated by textual replacement using the C preprocessor.

First we present the translation of data types and the generation of the standard functions (see § 4.1). In § 4.2 we show how user-defined functions are translated into SMV.

## 4.1. Types

The first step in the translation of data types is to check whether the data types are finite. A data type dt is finite, if it is equal to Bool, or if in its definition all types $\mathtt{typ}_{i_j}$ (see § 2.3) are finite and different from dt.

For finite data types the translation starts with the generation of the symbolic constants for each value of the type. For each constructor $\mathtt{C}_i$ we generate symbolic constants as the concatenation of the constructor name with all possible combinations of the arguments: if a constructor has no arguments, just the name of the constructor is used. All symbolic constants are then numbered.

To use the type in SMV declarations we define a symbolic name for the type by listing all possible values. For example, translating the data type definition of Alarm in § 2.2 results in:

```
#define FuelEng1 0
#define FuelEng2 1
#define TempEng1 2
#define TempEng2 3
#define PonR     4
#define Alarm { FuelEng1, FuelEng2,
                TempEng1, TempEng2,
                PonR }
```

**Constructors.** Constructor functions are translated into case splits on all possible argument values. For example, the constructor Fuel of Alarm is translated into:

```
#define Fuel(X0)
  case
    X0 = Eng1 : FuelEng1;
    X0 = Eng2 : FuelEng2;
  esac
```

In the case of multiple argument types the case constructs for each argument are nested into all cases of the previous arguments.

**Discriminators.** The generation of discriminator functions uses a case construct with a default case. The translated discriminator for Fuel looks as follows:

```
#define is_Fuel(x)
  case
    x = FuelEng1 : True;
    x = FuelEng2 : True;
    1 : False;
  esac
```

**Selectors.** For every argument of a constructor, one selector function is generated. Like discriminator functions, selector functions have one argument. Fuel is a constructor with a single argument, hence only one selector is generated. Selector names are generated schematically.

```
#define Fuel1(x)
  case
    x = FuelEng1 : Eng1;
    x = FuelEng2 : Eng2;
  esac
```

Using these definitions and the macro mechanism of the C preprocessor, we can translate expressions like

```
is_PonR(Fuel(Eng1)) = False
```

into SMV terms. Often, as in this example, the term consists only of constants and functions. Since SMV builds BDD-based normal forms, such terms are reduced to constant values, in this case to the value “true”. In our experiments so far, SMV had no difficulties parsing the input files. Of course, to reduce their size, another preprocessor could simplify such terms before passing them to SMV.

## 4.2. Functions

User-defined functions in AutoFocus are defined with pattern matching. A pattern is a variable or a constructor applied to further patterns. A term matches a pattern if it has the same constructors; variables in a pattern are bound to the matching subterms.

For the translation to SMV, constructor patterns are eliminated using discriminator and selector functions. Constructor functions remain only on the right hand side of a function definition.

The translation algorithm for pattern matching follows the pattern matching semantics given in the Haskell report [9]; here we only demonstrate the translation using the definition of the function `insertAlarm` (§ 2.3).

Assume the function is called as `insertAlarm(X1,X2)`. The function definition uses a pattern `Store(e1, e2, t1, t2, p)` for the second argument. In order for this pattern to match, the second argument must be of type Store. Therefore, in the translation the discriminator `is_Store(X2)` is introduced. Now, the variable `e1` must be matched with the first argument of the constructor, which is represented by the selector `Store1(X2)`. Variables match everything, so the condition “true” (in SMV: `1`) is generated, and the variable `e1` is bound to `Store1(X2)`. Thus, in the right hand side of the definition, each occurence of `e1` is replaced by `Store1(X2)`. Similarly, the remaining variables are matched and bound. In addition, the same strategy is used for the first argument.

The result of the translation of `insertAlarm` is shown in Figure 6.

## 5. Diagram Translation

Given the translation of data types and functions from § 4, it is straightforward to translate AutoFocus models into SMV. In this section, we describe how AutoFocus models are normalized prior to the code generation, how communication channels are encoded, how system structure diagrams are mapped into SMV modules, and finally how state transitions and state transition diagrams are translated.

### 5.1. Normalization

All AutoFocus description techniques support hierarchy as a means to reduce the complexity of specifications. For the SMV code, however, this hierarchy need not be preserved. In particular, it is sufficient to translate only those components into SMV that contain state transition diagrams. We refer to these components as *basic components*. Thus, before code generation we build a flattened AutoFocus model which consists only of basic components, and whose channels are derived from the transitive hull of the original system connection structure. To avoid name clashes in the flattened system structure, each component name is adorned with a unique number.

The state transition diagrams, too, are flattened. A flat STD is an STD where no state contains a sub-SSD. Note that a hierarchical STD can be regarded as a tree. The leaves of the tree are flat STDs, the inner STDs contain at least one state that holds a sub-STD. From this tree a new STD is built which contains all control states and state-to-state transitions of the tree’s leaf STDs. In addition, state-to-state transitions of the inner STDs are resolved by finding the corresponding connectors in the corresponding sub-STD and merging all connected transition segments. Again, control states need to be renamed to avoid name clashes. This is done by concatenating the names of the STDs on the path from root to the leaf STD, followed by the name of the control state in the leaf STD.

For example, in the normalization of the hierarchical STD of AlarmStore (Figure 4, 5), the states Do, Alarm, OK are renamed to ACDo, ACAlarm and ACOK, respectively. The transitions between the states in Figure 5 are all included. The transitions from the state NAV to AC in Figure 4 are linked with the corresponding transitions from the entry connection points in Figure 5; they now lead to states ACAlarm and ACOK.

The result of normalization is a flat SSD, where each component contains a flat STD with basic states only and transitions between them.

### 5.2. Channels

For the synchronous semantics of AutoFocus, each channel can carry at most one data value in each clock cycle. Hence simple variables suffice to hold the channel data. However, the presence or absence of a proper data value must also be encoded. Thus, each channel C is encoded as two variables: a value-carrying variable $C\_v$, and a Boolean variable $C\_p$ that denotes whether a proper value is present on the channel, or whether the channel is empty.

It is not really important where in the SMV code the channel variables are declared. We choose to declare each channel in the code of the STD that outputs into it.

### 5.3. System Structure

Each component of the flattened SSD is mapped into an SMV module. Each component module is parameterized with the value/presence variable pair for each component input channel. As mentioned above, output channels are declared locally in each component. Initially, all channels are empty: the presence variable of each output channel is set to false.

Below is the code fragment for the AlarmStore component. The parts marked by “. . . ” are produced by the STD translation, which is described in § 5.5.

```
MODULE AlarmStore_0(user_v, user_p, env_v, env_p)
  VAR out_v : Display;
  VAR out_p : boolean;
  ...
INIT
  !out_p & ...
TRANS
  ...
```

The type name `Display` is later replaced by integer ranges in a preprocessing step (see § 4.1).

The channels `Keys` and `Alarm` are inputs from the environment. They are declared in an abstract environment module, which is not present in the SSD but generated in the code generation:

```
MODULE Avionic_Env(show_v, show_p)
  VAR user_v : Keys;
  VAR user_p : boolean;
  VAR env_v : Alarm;
  VAR env_p : boolean;
INIT
        !user_p  & !env_p
TRANS   1
```

```
#define insertAlarm(X1,X2)
  case
    is_Fuel(X1) & is_Eng1(Fuel1(X1)) & is_Store(X2) & 1 & 1 & 1 & 1 & 1 & 1 :
         Store(True,Store2(X2),Store3(X2),Store4(X2),Store5(X2));
    is_Fuel(X1) & is_Eng2(Fuel1(X1)) & is_Store(X2) & 1 & 1 & 1 & 1 & 1 & 1 :
         Store(Store1(X2),True,Store3(X2),Store4(X2),Store5(X2));
    is_Temp(X1) & is_Eng1(Temp1(X1)) & is_Store(X2) & 1 & 1 & 1 & 1 & 1 & 1 :
         Store(Store1(X2),Store2(X2),True,Store4(X2),Store5(X2));
    is_Temp(X1) & is_Eng2(Temp1(X1)) & is_Store(X2) & 1 & 1 & 1 & 1 & 1 & 1 :
         Store(Store1(X2),Store2(X2),Store3(X2),True,Store5(X2));
    is_PonR(X1) & is_Store(X2) & 1 & 1 & 1 & 1 & 1 :
         Store(Store1(X2),Store2(X2),Store3(X2),Store4(X2),True);
  esac
```

**Figure 6. Translation of `insertAlarm`**

The transition relation of the environment is completely nondeterministic: The environment may send arbitrary key signals and alarms to the system in each cycle.

Finally, these modules are instantiated in the main module. The parameters of the individual modules are filled in with the variable names of the corresponding channels; the AlarmStore is instantiated as follows:

```
VAR AlarmStore_0 : AlarmStore_0(
      Avionic_Env.user_v, Avionic_Env.user_p,
      Avionic_Env.env_v,  Avionic_Env.env_p
    );
```

## 5.4. Transitions

The translation of transitions is similar to the translation of user-defined functions; in particular, pattern matching within the input patterns and transition variables are eliminated.

The result of the translation for the transition StoreFirstAlarm (§ 2.4) is:

```
env_p = 1           -- input

& isEmptyStore(as)  -- precondition

& next(out_p) = 1   -- output
& next(out_v) =
   Err(insertAlarm(env_v, as), es)

& next(last) =      -- actions
   env_v
& next(as) =
   insertAlarm(env_v, as)

& ControlState = csACOK
& next(ControlState) = csACAlarm
```

The variable names ending in "_p" and "_v" refer to the presence status and current value of communication channels, as mentioned in § 5.2. The last two lines represent the control state change of the transition; the state names are prefixed with AC because of the normalization step.

## 5.5. Behavior

In addition to the output channel variables, each component defines the following state variables:

- A control state variable (an enumeration type).
- The data state variables; like channel types, their types are later replaced by integer ranges in a preprocessing step.

Initially, the control state variable is set to the name of the STD's initial state; for the component AlarmStore this a state called Sector in NAV. The data variables are set to their initial values (see Figure 3). Again, the symbolic constants are later replaced by integer values in the preprocessing step.

```
MODULE AlarmStore_0(user_v, user_p, env_v, env_p)
  VAR out_v : Display;
  VAR out_p : boolean;
  VAR ControlState: {
           csNAVSector, csNAVNorth, csNAVRose,
           csACAlarm, csACDo, csACOK };
  VAR as : AlarmStore;
  VAR es : EngineState;
  VAR last : Alarm;
INIT
  !out_p &
  ControlState = csNAVSector &
  as = EmptyStore &  es = AllEngOp & last = None
TRANS
```

| | Components | Channels | Types | Constants | States | Transitions | Inputs | BDD nodes | Memory | Time |
|---|---|---|---|---|---|---|---|---|---|---|
| ALARM | 3 | 4 | 7 | 43 | 13 | 13 | 140 | 25287 | 3.6 MB | 1.8 s |
| SSB | 27 | 56 | 2 | 8 | 20 | 62 | 186624 | 12642 | 1.7 MB | 0.4 s |
| SSB2 | 27 | 56 | 3 | 10 | 22 | 90 | 470596 | 51339 | 4.6 MB | 2.5 s |
| SSBBig | 27 | 56 | 2 | 8 | 22 | 62 | 96 Mio | 12003 | 296 MB | 139 s |

**Figure 7. Verification Results**

```
  -- Transition relation:
  ...
|
  -- Idle transition:
  ...
```

The transition relation is expressed as a formula in disjunctive normal form. Each minterm is a single transition of the STD, as presented above.

The last transition is the idle transition, which is not explicitly visible in the AutoFocus model. Its purpose is to make the transition relation total. It is taken when no other transition is enabled. Thus its precondition is the negation of the conjunction of all other transition's preconditions and input statements; it leaves all data variables and the control state unchanged and clears all output channels.

## 6. Case Study: Storm Surge Barrier

The storm surge barrier in Oosterschelde prevents the Netherlands from catastrophic floods like the one in 1959, which took the lives of more than 1000 people. The barrier consists of several slides that separate the Eastern Scheldt (inside) from the North Sea (outside). The slides are closed, when the water level of the north sea is too high.

There is an emergency closing system that controls the slides. This system is currently redesigned, and due to the high criticality of the system the design has to be formally verified. One safety critical property is that the "open" signal is sent to the barrier when

- the "close" signal had been sent before, and
- the inside water level becomes higher than the outside water level (plus a fixed constant) for the first time after the "close" signal was true, and
- the operator did not forbid giving the "open" signal.

The AutoFocus model is a complete model of the system, according to the informal specification of the functions. It also includes redundancy for the water level sensors with a two-three majority vote. The only abstraction made was that we used discrete input values for the six sensor signals that measure the water levels. The model has been translated to SMV; the proof of the critical properties takes less than one second.

Figure 7 shows some results concerning the size of some variations of the case study. For each system, we measured the number of components, channels, data types; the number of values and constants produced by the data type translation, the number of states and transitions in the STDs, and the number of possible input combinations from the environment to the system (in one time interval). The other numbers describe the ressources used by SMV; the results were produced on a SUN Ultra 2.

The SSB2 system in the figure is a refinement of the storm surge barrier that allows the sensors to send errors to the system. It could be proved that the system behaves correctly, even if there are faulty input values. SSBBig is a system which allows more different input values for the water level sensor signals.

The first line of the table results from the alarm management system that we used throughout this paper; the property checked was a trivial consistency test.

Our experience from the storm surge barrier case study is that the AutoFocus model is adequate for the formalization of the system, and that the critical properties of the barrier could be verified using our translation approach and SMV. Moreover, the description techniques of AutoFocus posed no difficulties for the engineers working with us.

## 7. Conclusion and Further Work

The main goal of our work is to combine intuitive graphical description techniques for system structure and behavior and an expressive data type specification language with automatic verification techniques. Thus we hope to reduce the problem of the acceptability of formal models and to make verification more accessible to industrial software developers.

This paper is a first step towards this goal. We show how high-level specifications can be translated into the hardware-oriented language of SMV. To our knowledge, no other automatic verification system allows functional specification of data types. We believe this is a main feature of our approach, as pattern matching allows us to concisely formulate complex transition conditions and actions of transition diagrams. Without user-defined functions it would often be necessary to introduce a special component that computes a function's result. The system specification would then have to trigger this component and to wait for its result; this introduces lots of intermediate states and leads to cluttered and overly complicated specifications.

The translation of diagrammatic and functional specifications is part of a larger ongoing project: Within the project Quest [22, 20], we develop a tool set that consists of AutoFocus, the model checker SMV, a test environment as well as the theorem proving system VSE [15]. Moreover, using similar techniques as the ones presented in this paper, we can also directly generate production code from AutoFocus.

In particular, the theorem prover is essential for the verification of larger systems. Using AutoFocus it is easy to model systems that cannot be handled directly by SMV, especially when more complex or even recursive (i.e. infinite) data types are used. Using the Quest toolset, it is possible to verify a system by *abstraction*. Properties that are proven for a smaller system also hold in the original system, provided a number of abstraction conditions hold [5, 18]. These abstraction conditions are automatically generated by AutoFocus and fed to the VSE prover, where they can be discharged using interactive proof. The conditions are formulas in simple predicate logic, and much easier to prove than temporal logic formulas. Abstraction can be used to reduce the system SSBBig to SSB (Figure 7).

Another extension that is currently being integrated into AutoFocus is the visualization of the counter examples generated by SMV as extended event traces (EETs, [21]). EETs are a simple variant of message sequence charts; they are a diagrammatic notation for single system runs or groups of similar runs. EETs can also be used to drive the simulation facilities of AutoFocus, so that counter examples can be interactively replayed.

Finally, the data type translation techniques presented in this paper can be applied for other specification and programming notations; for example, the synchronous programming language Esterel [2] could be extended with a host language independent data type system.

## References

[1] J.-R. Abrial. *The B-Book: Assigning Programs to Meanings.* Cambridge University Press, 1996.

[2] G. Berry and G. Gonthier. The ESTEREL synchronous programming language: Design, semantics, implementation. *Science of Computer Programming*, 19:87–152, 1992.

[3] R. Bird and P. Wadler. *Introduction to Functional Programming.* Prentice-Hall, 1989.

[4] R. Breu, U. Hinkel, C. Hofmann, C. Klein, B. Paech, B. Rumpe, and V. Thurner. Towards a formalization of the Unified Modeling Language. In *Proceedings of ECOOP'97, LNCS 1357*, 1997.

[5] E. M. Clarke, O. Grumberg, and D. E. Long. Model checking and abstraction. *Proc. 19th ACM Symposium on Principles of Programming Languages*, January 1992.

[6] E. M. Clarke, O. Grumberg, and D. E. Long. Verification tools for finite-state concurrent systems. In *A Decade of Concurrency, LNCS 803*, pages 124–175, 1994.

[7] M. Eckrich, W. Mala, W. Damm, K. Winkelmann, M. Broy, and O. Slotosch. Korrekte Software für sicherheitskritische Systeme KORSYS. Statusseminar Softwaretechnologie 1998, Tagungsband, BMBF 1998, 1998.

[8] D. Harel. Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming*, 8:231 – 274, 1987.

[9] Report on the programming language Haskell 98. http://www.haskell.org/, February 1999.

[10] U. Hinkel. *Formale, semantische Fundierung und eine darauf abgestützte Verifikationsmethode für SDL.* PhD thesis, Technische Universität München, 1998.

[11] G. J. Holzmann. The Spin Model Checker. *IEEE Trans. on Software Engineering*, 23(5):279–295, May 1997.

[12] F. Huber, S. Molterer, A. Rausch, B. Schätz, M. Sihling, and O. Slotosch. Tool supported Specification and Simulation of Distributed Systems. In *International Symposium on Software Engineering for Parallel and Distributed Systems*, pages 155–164, 1998.

[13] F. Huber, S. Molterer, B. Schätz, O. Slotosch, and A. Vilbig. Traffic Lights - An AutoFocus Case Study. In *1998 International Conference on Application of Concurrency to System Design*, pages 282–294. IEEE Computer Society, 1998.

[14] F. Huber, B. Schätz, and G. Einert. Consistent graphical specification of distributed systems. In *FME '97: 4th International Symposium of Formal Methods Europe, LNCS 1313*, pages 122 – 141, 1997.

[15] D. Hutter, B. Langenstein, C. Sengler, J. Siekmann, W. Stephan, and A. Wolpers. Deduction in the Verification Support Environment(VSE). In *FME '96: Industrial Benefits and Advances in Formal Methods, LNCS 1051*, 1996.

[16] J. C. C. M. B. Dwyer, G. S. Avrunin. Property specification patterns for finite-state verification. In *2nd Workshop on Formal Methods in Software Practice*, 1998.

[17] K. L. McMillan. *Symbolic Model Checking.* Kluwer Academic Publishers, 1993.

[18] O. Müller. *A Verification Environment for I/O Automata Based on Formalized Meta-Theory.* PhD thesis, Technische Universität München, 1998.

[19] L. Paulson. *ML for the Working Programmer.* Cambridge University Press, 1991.

[20] Quest. `http://www4.in.tum.de/proj/quest`.

[21] B. Schätz, H. Hußmann, and M. Broy. Graphical development of consistent system specifications. In *FME'96: Industrial Benefit and Advances In Formal Methods, LNCS 1051*, 1996.

[22] O. Slotosch. Overview over the project Quest. In *FM-Trends '98, LNCS 1641.*

[23] J. M. Spivey. *Understanding Z: A Specification Language and its Formal Semantics*, volume 3 of *Cambridge Tracts in Theoretical Computer Science.* Cambridge University Press, 1988.

[24] The VIS Group. VIS: A system for verification and synthesis. In *CAV '96, LNCS 1102*, pages 428–432, 1996.