Science of Computer Programming 78 (2013) 2435–2452

Contents lists available at ScienceDirect

Science of Computer Programming

journal homepage: www.elsevier.com/locate/scico

Formalising and analysing the control software of the Compact Muon Solenoid Experiment at the Large Hadron Collider

Yi Ling Hwong [a], Jeroen J.A. Keiren [b], Vincent J.J. Kusters [a,c,*], Sander Leemans [a,b], Tim A.C. Willemse [b]

[a] CERN, European Organization for Nuclear Research, CH-1211 Geneva 23, Switzerland

[b] Department of Mathematics and Computer Science, Eindhoven University of Technology P.O. Box 513, 5600 MB Eindhoven, The Netherlands

[c] Institute of Theoretical Computer Science, ETH Zürich, CH-8092 Zürich, Switzerland

ARTICLE INFO

Article history:
Received 19 November 2011
Received in revised form 29 November 2012
Accepted 30 November 2012
Available online 9 January 2013

keywords:
Case study
Process algebra
SML
Bounded model checking
Model transformations

ABSTRACT

The control software of the CERN Compact Muon Solenoid experiment contains over 27 500 finite state machines. These state machines are organised hierarchically: commands are sent down the hierarchy and state changes are sent upwards. The sheer size of the system makes it virtually impossible to fully understand the details of its behaviour at the macro level. This is fuelled by unclarities that already exist at the micro level. We have solved the latter problem by formally describing the finite state machines in the mCRL2 process algebra. The translation has been implemented using the *ASF+SDF meta-environment*, and its correctness was assessed by means of simulations and visualisations of individual finite state machines and through formal verification of subsystems of the control software. Based on the formalised semantics of the finite state machines, we have developed dedicated tooling for checking properties that can be verified on finite state machines in isolation.

1. Introduction

The Large Hadron Collider (LHC) experiment at the European Organization for Nuclear Research (CERN) has been built in a tunnel 27 km in circumference and is designed to yield head-on collisions of two proton (or ion) beams of 7 TeV each. On 30 March 2010, it achieved its first successful 7 TeV collision, breaking its previous world record, setting a new one. The LHC will continue to operate at half energy until the end of 2012; it will not run at full energy, achieving 14 TeV collisions, until 2014.

The Compact Muon Solenoid (CMS) experiment is one of the four big experiments of the LHC. It is a general purpose detector to study the wide range of particles and phenomena produced in the high-energy collisions in the LHC. The CMS experiment is made up of 7 subdetectors, with each of them designed to stop, track or measure different particles emerging from the proton collisions.

The control, configuration, readout and monitoring of hardware devices and the detector status, in particular various kinds of environment variables such as temperature, humidity, high voltage, and low voltage, are carried out by the Detector Control System (DCS). The control software of the CMS detector is implemented with Siemens' commercial Supervision, Control And Data Acquisition (SCADA) package PVSS-II and CERN's Joint Controls Project (JCOP) framework

* Corresponding author at: CERN, European Organization for Nuclear Research, CH-1211 Geneva 23, Switzerland. Tel.: +41 44 632 97 27.
E-mail address: vincent.kusters@inf.ethz.ch (V.J.J. Kusters).

0167-6423/$ – see front matter © 2013 Elsevier B.V. All rights reserved.
doi:10.1016/j.scico.2012.11.009

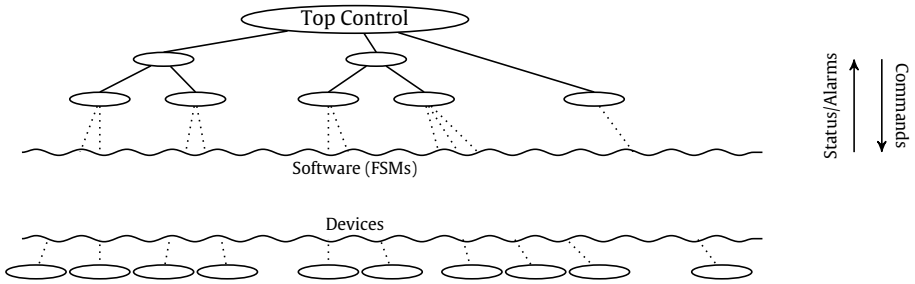

Fig. 1. Architecture of the real-time monitoring and control system of the CMS experiment, running at the LHC.

[1]. The architecture of the control software for all four big LHC experiments is based on the SMI++ framework [2,3]. Under the SMI++ framework, the real world is viewed as a collection of objects behaving as finite state machines (FSMs). These FSMs are described using the State Manager Language (SML), which can be seen as part of the SMI++ framework. SML is actively maintained and quite stable: new language features are only being added at a slow rate. However, its semantics is only defined informally.

A characteristic of the used architecture is the regularity and relatively low complexity of the individual FSMs and device drivers that together constitute the control software; the main source of complexity is in the cooperation of these FSMs. Cooperation is strictly hierarchical, consisting of several layers; see Fig. 1 for a schematic overview. The FSMs are organised in a tree structure where every node has one parent and zero or more children, except for the top node, which has no parent. Nodes communicate by sending commands to their children and state updates to their parents, so commands are refined and propagated down the hierarchy and status updates are sent upwards. Hardware devices are typically found only at the bottom-most layer. The only real time behaviour in the system is found in these device drivers. The average depth of the control tree is approximately 9 nodes, with a minimal depth of 3 nodes and a maximal depth of 11 nodes. The FSM system in the CMS experiment contains well over 25 000 nodes. The exact number fluctuates as a result of continuous development of the control system; a recent count revealed over 27 500 nodes. On average, each FSM contains 5 logical states (*i.e.*, control locations), and each logical state of an FSM can be in one of two phases; this would amount to at least $10^{27\,500}$ states for the entire configuration. While a large fraction of these states may not be reachable, we believe that $10^{27\,500}$ is still a very conservative estimate for the number of semantically reachable states (*i.e.*, states consisting of the control location and the valuations for local variables), as the logical states aggregate states that are semantically not related.

The sheer size of the system significantly contributes to its complexity. Complicating factors in understanding the behaviour of the system are the diversity in the development philosophies in subgroups responsible for controlling their own subdetectors, and the huge amount of parameters to be monitored. In view of this complexity, it is currently impossible to trace the root cause of problems when unexpected behaviours manifest themselves, since problems may have an effect on a large part of the hierarchy. A single badly designed FSM may be sufficient to lead to a livelock, resulting in non-responsive hardware devices, potentially ruining expensive and difficult experiments. Considering the scientific importance of these experiments, this justifies the use of rigorous methods for understanding and analysing the system.

Our contributions are twofold. First, we have formalised SML by mapping its language constructs onto constructs in the process algebraic language mCRL2 [4]. Second, based on our understanding of the semantics of SML, we have identified properties that can be verified for FSMs in isolation, and for which we have developed dedicated verification tooling.

Using the ASF+SDF meta-environment [5], we have developed a prototype translation implementing our mapping of SML to mCRL2. This allows us to quickly assess the adequacy of the translation through simulation and visualisation of FSMs in isolation, and by means of formal verification of small subsystems of the control software, using the mCRL2 toolset. The feedback obtained by the verification and simulation enables us to further improve the transformation. The use of the ASF+SDF meta-environment allows us to repeat this cycle in quick successions, and, at the same time, maintain a formal description of the translation. Development of the ASF+SDF Meta Environment was discontinued in 2010, when we had already started with our first experiments with ASF+SDF. Nevertheless, we chose to continue using ASF+SDF over similar products such as ATL because of our familiarity with ASF+SDF. Its syntax-driven, functional approach results in very clear translation rules. As a result, our translation can easily be converted to another formalism if needed.

The dedicated verification tools we developed allow engineers at CERN to quickly perform behavioural sanity checks on their design, and use the feedback of the tools to further improve on their designs in case of any problems. Results using these tools are favourable. For instance, our analysis reveals that 20% of the nodes in CMS, not counting the leaf nodes, can indeed suffer from such issues. That amounts to over 5% of all nodes. Moreover, actual outages of the control system have been traced back to livelocks found by our tools based on analysis of the logs. While our prototype tooling for verifying dedicated properties was built upon the ASF+SDF translation, we have recently reimplemented part of that translation in Python. The performance gains this brought about has allowed the engineers at CERN to integrate our dedicated tools in their development environment.

Related work. Popular verification techniques are theorem proving and model checking [6,7]. In this paper we focus on the latter approach. In model checking, models are represented as a finite state machine, and properties about these models

are expressed using temporal logic. Model checking research has spurred the creation of explicit state model checkers like CESAR [7], EMC [6,8], FDR [9], SPIN [10], and mCRL2 [4].

Explicit state model checking suffers from the infamous state space explosion problem; in parallel systems, the number of states in the complete system grows exponentially in the number of parallel components. Several techniques have been developed to tackle this explosion, *e.g.*, symbolic model checking [11], symmetry reduction [12,13] and compositional verification [14,15]. These techniques can indeed enable the verification of larger systems; [11] promises verification of "10^{20} states and beyond". However, from 10^{20} states there still is a long way to go before we can verify the $10^{27,500}$ estimated states in our system. We therefore think it is reasonable to assume that these techniques will only help to a limited extent, and that to verify the control software, we need to exploit the structural properties of the system.

The availability of state-of-the-art model checkers has enabled the verification of industrial systems. Among others the ISO/IEEE 1073.2 standard for remote control [16,17], the IEEE 1394 firewire standard [18–20], contention resolution in the ZigBee protocol [21], and the Carrier Sense Multiple Access/Collision Detection protocol in IEEE 802.3 [22] have been verified. However, as far as we are aware, there is no case study in the literature that parallels ours in terms of complexity.

The applicability of the model checker mCRL2 has been demonstrated in a number of industrial case studies. Among others an automated parking garage [23], a distributed system for lifting trucks [24], part of the IEEE 11073-20601 protocol for personal health devices [25], a distributed grid system supporting production activities as well as user data analysis used at CERN [26], as well as a printer intended to operate in the manufacturing of printed circuit boards [27] have been verified.

Outline. We give a cursory overview of the core constructs of the SML language and the configuration of the control system in Section 2. In Section 3, the mCRL2 language is explained. The mCRL2 semantics of SML are then explained in Section 4, and we briefly elaborate on the methodology we used for obtaining this semantics. Our dedicated verification tools for SML, together with the results obtained so far, are described in further detail in Section 5. We summarise our findings and suggestions in Section 6.

2. The control system

2.1. The control system configuration

The CMS control system is organised as a tree-like architecture (see also Fig. 1). Nodes in this tree are the basic behavioural entities, sending commands down the tree to their children and state updates up the tree to their parents. Communication between the nodes is arranged through a proprietary middleware layer, called *DIM* [28], which is based on the client/server paradigm, offering reliable asynchronous communication between nodes.

The tree-like configuration and the finite state machine descriptions of the CMS control system are stored in a database for easy maintenance. That is, for each node, it is precisely known which node is its parent and which nodes are its children, and the SML code that dictates the behaviour of the node.

2.2. The State Manager Language

The finite state machines used in the CMS experiment are described in the State Manager Language (SML) [2,3]. In *ibid.* only an informal semantics is given, and a formal semantics of SML is unavailable. We present the syntax and the suggested meaning of the core of the language using snapshots of a running example; we revisit this example in our formalisation in Section 4, where also the abbreviations that are used in the formalisation are explained in more detail. Note that SML is larger than presented here, also offering language constructs for the hardware device drivers, but the FSMs making up the control system employ these core constructs only.

Listing 1 shows part of the definition of a *class* in SML. Conceptually, this is the same kind of class known from object-oriented programming: the class is defined once, but can be instantiated many times. An instantiation is referred to as a Finite State Machine. A class consists of one or more *state clauses*; Listing 1 only shows the state clause for the `OFF` state. Intuitively, a state clause describes how the FSM should behave when it is in a particular state. Every state clause consists of a list of *when clauses* and a list of *action clauses*, either of which may be empty.

A *when clause* has two parts: a *guard* which is a Boolean expression over the states of the children of the FSM and a *referrer* which describes what should happen if the guard evaluates to true. The base form of a guard is `P in_state STATE`, where `STATE` is the name of a state (or a set of state names) and `P` is a *child pattern*. A child pattern consists of two parts: the first part is either `ANY` or `ALL` and the second part is the name of a class or the literal `FwCHILDREN`. The intended meaning is straightforward:

```
$ALL$FwCHILDREN in_state ON
```

means "all children are in the `ON` state", and:

```
$ANY$RPC_HV in_state {RAMPING_UP, RAMPING_DOWN}
```

evaluates to true if "some child of class `RPC_HV` is either in state `RAMPING_UP` or state `RAMPING_DOWN`".

```
class: $FWPART_$TOP$RPC_Chamber_CLASS
    state: OFF
        when ( ( $ANY$FwCHILDREN in_state ERROR ) or
               ( $ANY$FwCHILDREN in_state TRIPPED ) )  move_to ERROR

        when ( $ANY$RPC_HV in_state {RAMPING_UP,
                                     RAMPING_DOWN} ) move_to RAMPING
        when ( ( $ALL$RPC_LV in_state ON ) and
               ( $ALL$RPC_HV in_state STANDBY ) )  move_to STANDBY

        when ( ( $ALL$RPC_HV in_state ON ) and
               ( $ALL$RPC_LV in_state ON ) )  move_to ON

        when ( ( $ALL$FwCHILDREN in_state ON ) and
               ( $ALL$RPC_T in_state OK ) )  move_to ON

        action: STANDBY
            do STANDBY $ALL$RPC_HV
            do ON $ALL$RPC_LV
        action: OFF
            do OFF $ALL$FwCHILDREN
        action: ON
            do ON $ALL$FwCHILDREN
```

Listing 1: Part of the definition of the *Chamber* class in SML.

A referrer is either of the form `move_to STATE`, indicating that the finite state machine changes its state to `STATE`, or of the form `do A`, indicating that the action with name `A` should be executed next. If the guards of more than one *when clause* evaluate to true, the topmost enabled referrer is executed. Whenever the FSM moves to a new state, it executes the *when clauses*, starting from the top *when clause*, to see if it should stay in this state (all guards are false) or if it should go to another state (some guard is true). It is therefore possible that a single `move_to` referrer or statement (see below) triggers a series of state changes.

An *action clause* consists of a *name* and a list of *statements*. When an FSM receives a command while in a state `STATE`, it looks inside the state clause of state `STATE` for an *action clause* with the same name as the command and if such an *action clause* exists, it executes its statement list. If no such action exists, the command is ignored. For example, if the *Chamber* finite state machine from Listing 1 is in state `OFF` and it receives an `ON` command, it will execute the last *action clause*.

The most commonly used statement is `do C P`, which means that the command `C` is sent to all children which match the child pattern `P`. After a command is sent, the child is marked *busy*. When a child sends its new state back, this *busy* flag is removed. The `do` statement is non-blocking, *i.e.*, it does not wait for the children to respond with their new state. The child pattern always starts with `$ALL$` in this context. SML also provides `if` and `move_to` statements, as we illustrate in Listing 2.

```
action: STANDBY
    do STANDBY $ALL$RPC_HV
    do ON $ALL$RPC_LV
    if ( $ALL$RPC_LV in_state ON ) then
        do ON $ALL$RPC_HV
        if ( $ALL$RPC_HV in_state ON ) then
            move_to ON
        endif
    else
        do STANDBY $ALL$RPC_LV
        do STANDBY $ALL$RPC_HV
        do STANDBY $ALL$FwCHILDREN
    endif
```

Listing 2: An example of a more complex *action clause*.

The `move_to STATE` statement immediately stops execution of the *action clause* and causes the FSM to move to `STATE`. The `if G then S1 else S2 endif` statement blocks as long as there is a child, referred to in `G`, that has a busy flag. If the guard `G` evaluates to true, then `S1` is executed and otherwise `S2` is executed. The else clause is optional.

3. An overview of mCRL2

The mCRL2 language [4] consists of three distinct parts: a *data language* for describing the data types and transformations, a *process language* for specifying system behaviours and a *modal language* for reasoning about the system behaviours.

3.1. Data language

The data language of the mCRL2 language is based on higher-order abstract data types, in which functions are defined using equational specifications. The language has built-in definitions for many of the commonly used data types (and operations on them), such as Booleans, represented by sort `Bool`, (unbounded) integer, natural and positive numbers, represented by sorts `Int`, `Nat` and `Pos`, respectively. Container sorts, such as (infinite) sets, bags and lists over arbitrary data sorts `D` are denoted by `Set(D)`, `Bag(D)`, and `List(D)`. New data sorts can be defined either by directly specifying the constructors of a data sort, or using structured sorts, or through aliasing. Relations and mappings on data sorts, and their rules of logic are formalised through equational rewrite rules. Defining data sorts through structured sorts introduces a few built-in mappings and relations, such as projection functions, equality and inequality. Expressions in the data language can be built by combining sort constructors, functions, relations and data variables. Example expressions are `3+4` of sort `Pos`, and `b && (b || false)` and `n == 10` of sort `Bool`.

Example 1. A sort representing the Cartesian product of integer numbers can be defined by aliasing it to a structured sort `struct coord(x:Int,y:Int)`, *i.e.*, `Point = struct coord(x:Int,y:Int)`. The mappings `x` and `y` are the projection functions: given an expression `coord(0,1)` of sort `Point`, the expression `x(coord(0,1))` is an integer number (the number `0` in this case).

Of particular interest for the formalisation of SML programs is lambda abstraction, which is well-known from λ-calculus and type theory. In our specification, we use them, *e.g.*, to model association lists. Functions, specified through lambda abstraction or otherwise, can be updated concisely and intuitively, as illustrated by the following example.

Example 2. Let `f:Nat->Nat` be a user-defined mapping of sort `Nat` to `Nat`; the function `f` can be thought of as an infinite array. Using lambda abstraction, we can define `f` to map each natural number to its doubled number: `f=lambda n:Nat. n+n`. Note that this is equivalent to defining `f(n) = n + n`, but allows inline definition of this function. Using function updates, function `f` can be modified: `f[0->2]` agrees with function `f`, save for `f[0->2](0)`, which has value `2`, whereas `f(0)` has value `0`.

The built-in data types are designed to reflect their mathematical counterparts, contributing to the accessibility of the data language. The support for universal and existential quantifiers further facilitates conventional mathematical reasoning, but since we do not use them in our formalisation, we refrain from an in-depth explanation.

3.2. Process language

The process specification language of mCRL2 consists of only a small number of basic operators and primitives. The language is inspired by process algebras from the ACP family [29], and has both an axiomatic and an operational semantics. We forego a formal exposition of its semantics, for which we refer to [4,29]; instead, we restrict ourselves to introducing its syntax, sketch its meaning informally and illustrate its use through small examples.

Processes are constructed compositionally using alternative composition and quantification, sequential and parallel composition, hiding, communication, and recursion. The basic behavioural elements of a process are the deadlock process `delta` and (parameterised) actions. The latter represent atomic, observable events, such as receiving status updates or the sending of commands; the parameters can be used to represent the data that is linked to such events. If `read` is an action name, and `e` is some data expression of the sort of action name `read`, then `read(e)` is a parameterised action.

Suppose `p` and `q` are processes, then their alternative composition is denoted `p+q`. Intuitively, process `p+q` behaves as either process `p` or `q`, dependent on which of the two processes executed the first action. Since process `delta` can never execute actions, process `delta+p` will simply behave as process `p`. Alternative quantification generalises alternative composition: if a data variable `d` of some sort `D` occurs in process `p`, then `sum d:D. p` denotes the (possibly infinite) choice between the set of processes obtained by instantiating variable `d` in process `p` with all possible values it can attain.

Example 3. Suppose `read(n+n)` is an action parameterised with natural number expressions, then process `sum n:Nat. read(n+n)` denotes the infinite set of processes offering `read` actions with even values as their parameters.

Using a binary *if-then* construct, denoted `b -> p`, or a ternary *if-then-else* construct, denoted `b -> p <> q`, processes can be made data-dependent: if `b` evaluates to `true`, then processes `b -> p` and `b -> p <> q` behave as process `p`, and the latter behaves as process `q` otherwise, whereas the former behaves as process `delta`.

The sequential composition of processes `p` and `q`, denoted `p.q` behaves as process `p`, and, upon successful termination of `p`, continues to behave as process `q`. Note that the deadlock process `delta` never terminates successfully; hence, process `delta.p` is indistinguishable from process `delta`.

Example 4. Let `read` and `send` be two parameterised actions, taking natural number expressions as their arguments, and let `n` of sort `Nat` be a data variable. Then process `read(n).send(n)` represents a system in which first a value represented by data variable `n` is read, which is then sent via action `send`. The process `sum n:Nat. read(n).send(n)` combines sequential composition with alternative quantification, modelling that any natural number read through action `read` is sent via action `send`.

A parallel composition of processes p and q, which is denoted by process p||q, behaves as the interleaving of both processes involved: the first action may come from process p, which after execution of this action behaves as process p'; the resulting process then is p'||q. Symmetrically, the first action may come from process q. In addition, both processes may execute their first actions simultaneously, producing a *multi-action*, after which the processes that remain are again composed in parallel.

Example 5. Consider the process p defined as `sum n:Nat. read(n).send(n)` and q, defined as `sum m:Nat. send(m).read(m+m)`. The parallel composition of p and q, may first execute a `read(n)` action, after which it will behave as the parallel composition `send(n)||q`, it may first execute a `send(m)` action, after which it will behave as the parallel composition `p||read(m+m)`, *or* both processes may execute their first actions simultaneously, denoted by the multi-action `read(n)|send(m)`, after which the remaining process behaves as `send(n)||read(m+m)`.

Exchanging information between processes by synchronising on specific events is achieved using the binary `comm` operator. This operator takes a set of communication rules and a process as its argument; the communication rules specify which multi-actions communicate successfully. Exchange of information and successful communications are only achieved if the involved actions agree on all the values of their parameters.

Example 6. Consider again processes p and q from the previous example. Suppose actions `read` and `send` can communicate, yielding a new parameterised action `sync`. Then process `comm({read|send->sync}, p||q)` will convert the multi-action `read(n)|send(m)` to `sync(n)` whenever `n==m`, and will leave the multi-action intact in all other cases.

Process behaviours can be restricted to only use a specific set of atomic actions and multi-actions, using the binary `allow` construct. This provides the means to enforce that non-successful synchronisations of two parallel processes are not considered. Effectively, the process `allow(A,p)`, where A is a finite set of atomic action names and multi-actions, behaves as process p, except that any action or multi-action in p that is not in A is replaced by the deadlock process `delta`.

Example 7. Let p and q again be the processes from the previous example. The process `allow({sync}, comm({read|send->sync}, p||q))` will behave as `sum n:Nat. sync(n).( sum m:Nat. (n==m+m) -> sync(n) <> delta)`. That is, if a non-zero value is communicated between processes p and q, then no further communication happens (since `n==m+m` evaluates to `true` only if both n and m are zero) and the process locks; if the value zero is exchanged, then this is done once more, after which the process successfully terminates.

Finally, recursive equations of the form $X(d_1:D_1,\ldots,d_n:D_n) = p$, allow for specifying infinite behaviours. Intuitively, each occurrence of a parameterised process variable $X(e_1,\ldots,e_n)$ in some process term q behaves as process p, in which the variables d_i have been replaced by expressions e_i.

Example 8. Consider process equation `X=sum m:Nat. read(m).X` and process equation `Y(n:Nat)=sum i:Nat. (i > n) -> send(i).Y(i)`. Process X can execute a `read` action with an arbitrary natural number parameter, after which it again behaves as process X. Process `Y(0)` can execute a `send` action with an arbitrary non-zero natural number parameter `i`, after which it behaves as process `Y(i)`. Effectively, process `Y(i)` specifies a system that sends ever-increasing numbers.

If a process variable occurs within the scope of its own equation, a shorthand notation for updating only part of the data parameters of that equation is available. For instance, if process variable $X(d_1,\ldots,e_i,\ldots,d_n)$ occurs in the right-hand side process p of an equation $X(d_1:D_1,\ldots,d_n:D_n) = p$, then we can write $X(d_i=e_i)$ instead.

Example 9. Consider the process equation `X(n:Nat, m:Nat) = sum p:Nat. read(p).X(n, p)` in which only m is changed in the right hand side. Using our shorthand notation, we can also write process equation `X(n:Nat, m:Nat) = sum p:Nat. read(p).X(m=p)`.

3.3. Modal language

Whereas the process language is typically used to specify how a system achieves its behaviour, the modal language is typically used to reason about high level requirements of such systems. The modal language of mCRL2 is based on the theory of the modal μ-calculus [30], extended with facilities to reason about data, see [31,32]. The resulting language is quite expressive; for instance, it admits a linear encoding of the temporal logic LTL, see [33].

Apart from standard Boolean connectives such as conjunction and disjunction, the mCRL2 modal language permits the use of existential and universal quantification over data sorts (specified by the data language), and the use of Boolean expressions. In addition the language permits the use of modalities. The *must* modality `[A]f` expresses that any first action `a(v)` executed by a process will result in a process that satisfies property `f` if action `a(v)` is among the actions in the set of actions described by A. Dually, the *may* modality `<A>f` asserts that among the set of first actions that can be executed by the process, there is one action that is contained in the set of actions described by A, and which, if executed, will result in a process satisfying property `f`. The modal language permits describing infinite sets of actions, which is needed because of the possibly infinite branching processes that can be described by the process language.

Example 10. The set of actions characterised by `true` is the entire set of actions; the set of actions characterised by `exists n:Nat. read(n)` is the set of read actions with a parameter taken from the set of all natural numbers. Finally, `!exists n:Nat. read(n+n)` specifies the entire set of actions except those read actions with even valued natural number parameters, *i.e.* the `!` operator denotes the complement of a set of actions. The example given here contains, *e.g.*, the actions `a`, `send(m)` and `read(1)`, but not `read(2)`. Thus, `<true>true` asserts that a process *can* execute an action, and `[!exists n:Nat. read(n+n)]false` asserts that a process can at most execute read actions with even valued natural numbers. Lastly, `forall n:Nat. <read(n)>true` asserts that a process can execute read actions with every natural number parameter. This property holds for a process such as `sum m:Nat. read(m)`, but not for a process such as `sum m:Nat. read(m+m)`, since the latter cannot perform, *e.g.*, a `read(1)` action.

Finally, least and greatest fixpoints, denoted by `mu X. f(X)` and `nu X. f(X)`, respectively, permit reasoning about finite and infinite runs of a system. Typically, least fixpoints are used to specify *eventualities*, whereas greatest fixpoints are used for *invariants*. By mixing least and greatest fixpoints, increasingly complex properties, such as fairness properties, can be stated. We refer the reader to [34] for an excellent in-depth discussion of the μ-calculus. In this paper we consider worst-case behaviour of the system, hence we do not use fairness in any of our properties.

Example 11. The property `nu X. <exists n:Nat. read(n)>X` asserts that a process is capable of executing an infinite sequence of read actions, without requiring anything about the parameters these carry. In a similar vein, the property `forall n:Nat. [read(n)]mu X. ([!send(n)]X && <true>true)` asserts that a read action with some natural number parameter will inevitably result in a send action with that same natural number parameter.

The modal language of mCRL2 features *parameterised* recursion, but since we will not use this construct, we will not elaborate on it.

3.4. Tooling

The language mCRL2 has a homonymously named toolset, offering tools that help to understand specifications written in the data language and the process language, and tools that can check whether properties written in the modal language hold of a process description or not. For an overview of most common tools, we refer to [35]; we here confine ourselves to give a high-level overview of the techniques that were most relevant for our purposes.

Data expressions can be evaluated using an interpreter. Typically, the interpreter can help in understanding why expressions can be further simplified or not. For instance, one may ask whether for a complex open Boolean data expression `b`, there is an assignment to some of the variables in `b` so that the expression evaluates to `true`. Note that it is very easy to state undecidable properties in the language, so such tooling helps assess whether the technology used for reasoning about data is sufficiently powerful to work with the expressions used, *e.g.*, in the process description of a certain system. For instance, one may wonder whether for an unknown natural number `k`, the expression `exists n:Nat. k < n && n < 4` evaluates to `true`, `false`, or some other expression; in this case, the interpreter will simplify the expression to `k < 3`.

The behaviour described by processes can be simulated or explored exhaustively by investigating the combinatorial possibilities of the actions that can be executed, resulting in a *state space* such as a labelled transition system. Such a state space can be visualised in 2D or 3D using a variety of advanced techniques. Moreover, reduction techniques allow for minimising the state space using well-known equivalence reductions such as strong bisimilarity, similarity, trace equivalence and (divergence-sensitive) branching bisimilarity.

Verification of the behaviour described by processes is supported by computing whether a given functional requirement, expressed as a modal μ-calculus formula holds for the process or not; this is known as *model checking*. For specific types of requirements, counterexamples that are easy to interpret can be reported in the case the requirement fails on the given process. This facilitates debugging the cause of the failure.

4. A formal semantics for SML

We next describe the most important aspects of the translation of SML to mCRL2. The details of the formal translation of SML into mCRL2 can be found in the digital appendix. Our choice for mCRL2 is motivated largely by the expressive power of the language, its rich data language rooted in the theory of abstract data types, its available tool support, and our understanding of the advantages and disadvantages of mCRL2.

4.1. From SML to mCRL2

Every SML class is converted to an mCRL2 process definition; the behaviour of an FSM is then described by the behaviour of a process instance. Each FSM maintains a state and a pointer to the code it is currently executing. In addition, an FSM is embedded in a global tree-like *configuration* that identifies its parent, and its children. In order to faithfully describe the behaviour of an FSM, we therefore equip each mCRL2 process definition for a class `X` with this information as follows:

```
proc X_CLASS(c: Config, s: State, cs: ID -> State, busy: Children
             phase: Phase, cq: CommandQueue, pc: Int)
```

Parameter c represents the static configuration of a process instance. This static configuration is of sort Config, a structured sort:

```
Config = struct configuration(self: Id,
                              parent: Id,
                              chs: Children,
                              cc: Class -> Children);
Children = List(Id);
```

The structured sort Config can be thought of as a named tuple; self represents the unique identifier of this process instance, parent represents the unique identifier of the parent of this process instance, *i.e.*, its parent in the tree structure. The list of identifiers of the children of the node in the tree is contained in chs. An association list, mapping FSM classes to all children of said class is represented by cc.

Process parameter s is used to keep track of the state of the FSM. The state information of self(c)'s children is stored in cs, which is a function from child identifiers to states. Whenever the FSM receives a state-update message from one of its children, the function cs is updated accordingly. Children that are still processing the last command sent to them are contained in busy. A child is added to busy after sending a message to the child, and is removed when it responds with its new state.

The phase parameter has value WhenPhase if the FSM is executing the *when clauses* and ActionPhase otherwise; Phase is a simple structured sort containing these two values. The phases will be explained in detail in the following section.

We only need parameters cq and pc in the *action phase*. The command queue cq contains messages that are to be sent to an FSM's children. Specifically, when executing a do C P statement, we add a pair with the child's id and the command C to cq, for every child matching the child pattern P. The command queue is subsequently emptied by sending the messages stored in cq. We postpone discussion of the program counter represented by pc to the following section.

Guards in FSMs are described using a three valued logic, taking values TRUE, FALSE, and GHOST. Guards can test whether all or any of an FSMs children of a certain class are in a designated state. Formally, such a test is described as ALLP in_state STATE where P describes a set of children. This checks whether all children of which the class type is in P are in state STATE. Likewise, we can write ANYP in_state STATE. Recently, the language was extended to support for the check P is_empty, which can be used to test whether an FSM has no children with a class type matching P. Since this feature is currently not used in any FSM in production, we refrain from further discussion of the construct. Complex guards can be obtained by composing guards using operators and, or and not, and by testing for sets of states instead of a single state.

The logic used to evaluate guards has been constructed from an engineering perspective. If an ANY or ALL test refers to an empty set of objects, it is evaluated to GHOST; otherwise it returns a logical value that corresponds to the statement, *i.e.*, FALSE or TRUE. In a bigger context, the GHOST value is treated such that it is ignored in this context. This corresponds to the intuition that a condition about children that are absent should be ignored. While this may seem odd at first glance, this permits specifying FSMs that can function in different configurations, enabling a form of reuse of the FSM.

Example 12. Consider an FSM in which the following expression occurs within a guard:

```
( $ALL$RPC_LV in_state ON ) and ( $ANY$RPC_HV in_state ON )
```

If, in the configuration in which the FSM is used, the FSM has children of both classes RPC_HV and RPC_LV, then this guard evaluates to TRUE or FALSE, depending on the state of the children: if all children of class RPC_LV are in state ON and some child of class RPC_HV is in state ON, then the guard evaluates to TRUE; it evaluates to FALSE otherwise. However, the FSM may also be used in a configuration in which its children are only of class RPC_LV. Adopting standard two valued Boolean logic would evaluate (ANYRPC_HV in_state ON) to FALSE in this case. As a result, the entire guard would evaluate to FALSE, too. Now, consider the following expression, which is the disjunction of the same two clauses:

```
( $ALL$RPC_LV in_state ON ) or ( $ANY$RPC_HV in_state ON )
```

In this case, the presence of the test on the states of children of class RPC_HV does not influence the value of the guard. The developers of the language considered this difference to be undesirable; indeed, if the behaviour of an FSM is to only depend on statements made about children that are present, the quantification over RPC_HV children should not influence the value of the above expressions in case no children of class RPC_HV are present.

Naively extending Boolean logic with GHOST can easily lead to inconsistent theories in which one can derive true=false. Therefore, some care is to be taken when implementing this logic. In practice, the logic is implemented in several steps. First, all tests, such as ANY and ALL, are evaluated to FALSE, GHOST or TRUE. The resulting (ground) expression is subsequently rewritten to an expression containing at most one GHOST value. This is done using the following rewrite rules (here $x \in$ {FALSE, GHOST, TRUE} is an arbitrary three valued logic value):

$$\begin{array}{llll} x \text{ and GHOST} & \to x & \text{GHOST and } x & \to x \\ x \text{ or GHOST} & \to x & \text{GHOST or } x & \to x \\ \text{not GHOST} & \to \text{GHOST} & & \end{array}$$

As the penultimate step, every occurrence of the value GHOST is replaced with the value FALSE. Note that this step is only to be carried out *after* rewriting the expression using the above rewrite rules. Finally, the resulting expression is evaluated using standard rules of Boolean logic. Expressions that reduce to FALSE this way are interpreted as the mCRL2 Boolean value `false` and expressions that reduce to TRUE are interpreted as the Boolean value `true`.

Example 13. Reconsider the first expression from Example 12, *i.e.*:

```
( $ALL$RPC_LV in_state ON ) and ( $ANY$RPC_HV in_state ON )
```

Assume again that the FSM in which the expression occurs is used in a configuration with children of class RPC_LV, but no children of class RPC_HV. Suppose that all children of class RPC_LV are in state ON. Following the steps outlined above, first all tests such as ANY and ALL are evaluated to TRUE, FALSE or GHOST. This means that the expression of Example 12 leads to the following (ground) expression:

```
( TRUE ) and ( GHOST )
```

Next, the above expression is reduced using the rewrite rules outlined above, yielding expression:

```
( TRUE )
```

Since no GHOST value remains, the resulting expression TRUE can be interpreted as a standard expression over standard Boolean logic, yielding the value `true`. Now consider the second expression of Example 12, *i.e.*:

```
( $ALL$RPC_LV in_state ON ) or ( $ANY$RPC_HV in_state ON )
```

Following the same recipe, we can also reduce this expression to `true`. In a sense, the expressions are rewritten by ignoring requirements on the non-existent children, which is what the engineers were after.

Note that following different algorithms may lead to different results. For instance, immediately replacing GHOST with value FALSE ultimately yields different values for the expressions of Example 12.

The logic is implemented as the mCRL2 sort `ThreeValuedLogic` of which the full implementation can be found in the digital appendix. An example of the translation of a guard to mCRL2 is shown in Translation 1; the function `bool` that is used in the transformation formalises the final step of the conversion of three valued logic (ground) expressions to an mCRL2 Boolean value.

SML

```
( $ALL$FwCHILDREN
    not_in_state OFF )
and
( $ANY$FwCHILDREN
    in_state STANDBY )
```

mCRL2

```
bool(
  and(
    all_not_in_state(chs(c), cs, [S_OFF]),
    any_in_state(chs(c), cs, [S_STANDBY])
  )
)
```

Translation 1: Translation of a condition.

4.1.1. Phases

During the *when phase*, a process executes *when clauses* until it reaches a state in which none of the guards evaluates to true. It then moves to the *action phase*. In the *action phase*, a process can receive a command from its parent or a state-update message from one of its children. This process is illustrated in Fig. 2. After handling the command or message, it returns to the *when phase*.

Translating the *when phase* turns out to be rather straightforward: for each state a process term consisting of nested if-then-else statements is introduced, formalised by mCRL2 expressions of the form `b->p<>q` (if b, then act as process p, otherwise as q). Each if-clause represents exactly one *when clause*. The else-clause of the last *when clause* sends a state-update message (represented by the mCRL2 action `send_state`) with the current state to the parent of this FSM and moves to the *action phase*. An example is given in Translation 2.

The `move_state` action indicates that the process changes its state. Note that for every `move_state` action that happens the state is sent to the parent through a `send_state` action. If none of the guards are true, the current state is sent to the parent and the process changes to the *action phase*, signalled by a `move_phase` action.

Modelling the *action phase* is more involved as we need a separate process to take care of sending commands to children. We focus on the translation of the *action clauses* and the code which handles state-update messages.

SML allows for an arbitrary number of statements and an arbitrary number of (nested) if-statements in every *action clause*. We uniquely identify the translation of every statement with an integer label. After executing a statement, the program counter `pc` is set to the label of the statement which should be executed next. There are two special cases here:

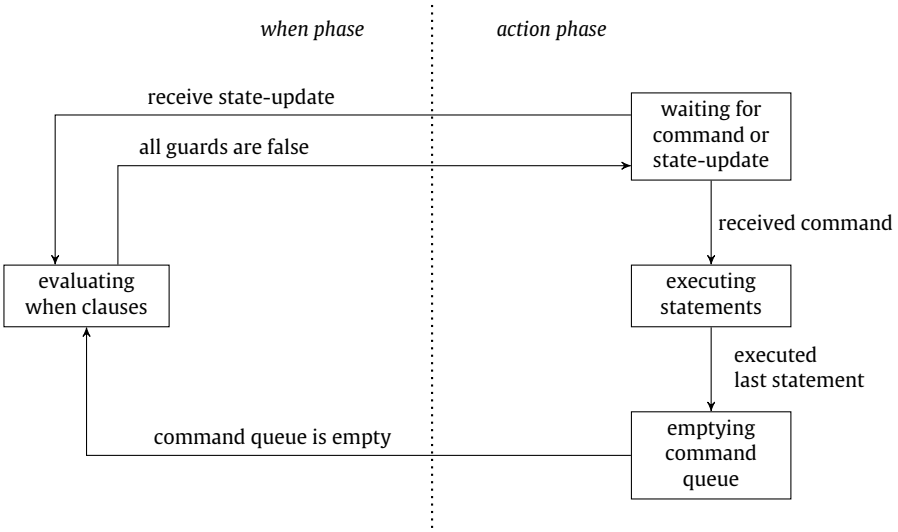

Fig. 2. Overview of the *when phase* and the *action phase*.

SML

```
state: OFF
  when G1 move_to S1
  ...
  when Gn move_to Sn
```

mCRL2

```
isS_OFF(s) && isWhenPhase(phase) -> (
  translation_of_G1 ->
    send_state(self(c),parent(c),S1).
    move_state(self(c),S1).
    X_CLASS(s = S1) <>
  ...
  translation_of_Gn ->
     send_state(self(c),parent(c),Sn).
     move_state(self(c),Sn).
     X_CLASS(s = Sn) <>
  send_state(self(c),parent(c),s).
  move_phase(self(c),ActionPhase).
  X_CLASS(phase = ActionPhase,
          cq = [], pc = 0))
```

Translation 2: Simplified translation of the *when clauses* of a state OFF.

- Label 0, the clause selector. When entering the *action phase*, the program counter is set to 0. Upon receiving a command, the clause selector sets the program counter to the label of the first statement of the *action clause* that should handle the command.
- Label -1, end of action. After executing an action, the program counter is set to -1, signalling that the command queue must be emptied and the process must change to the *when phase*.

An example is given in Translation 3. The `receive_command` action models the reception of a command that was sent by the FSM's parent. Such a command is ignored if no *action clause* handles it. In the example, observe that both after ignoring a command and after completing the execution of the `STANDBY` action handler, the program counter is set to -1. A process term not shown here then empties the command queue by issuing a sequence of `send_command` actions, and subsequently returns to the *when phase*.

Since a `do` statement is asynchronous, the children can send their state-update at any time during the *action phase*. This is dealt with as follows. Suppose a state-update message is received through the `receive_state` action. If this precedes the reception of a command in this *action phase*, we simply process the state-update and move to the *when phase*. If we are in the middle of executing an *action clause*, we process the state-update, but do not move to the *when clause*. The rationale is that sending commands to children is done instantaneously, and is, to all intents and purposes, atomic.

4.1.2. Formalising the hierarchy

So far we have only discussed the translation of individual FSM classes. The instantiation and integration of these classes, such that our translation describes the complete hierarchy of FSMs is done at a higher level in the specification. For each

SML

```
state: OFF
  action: STANDBY
    do STANDBY $ALL$Y
    do ON $ALL$Z
  action: OFF
    do OFF $ALL$Y
  action: ON
    do ON $ALL$Y
```

mCRL2

```
isS_OFF(s) && isActPhase(phase)
     && cq == [] -> (
  pc == 0 ->
    sum com:Command.(
      receive_command(parent(c),self(c),com).
      isC_STANDBY(com) ->
        X_CLASS(pc = 1) <>
      isC_OFF(com) ->
        X_CLASS(pc = 3) <>
      isC_ON(com) ->
        X_CLASS(pc = 4) <>
      send_state(self(c),parent(c),s).
      ignored_command(self(c),com).
      X_CLASS(pc = -1) +

  pc == 1 ->
    X_CLASS(
      cq = send_command(C_STANDBY,
             (cc(c))(Y_CLASS)),
      pc = 2) +

  pc == 2 ->
    X_CLASS(
      cq = send_command(C_ON,
             (cc(c))(Z_CLASS)),
      pc = -1) + ...
```

Translation 3: Simplified translation of the *action clauses* of a state OFF.

FSM, the process describing its class is instantiated with its id, the id of the parent, the ids, states and types of its children, and the parameters for the initialisation phase. The processes obtained in this way, are then put in parallel. Information about actual configurations are read from the FSM database.

The `send_state` action communicates with the `receive_state` action to a `comm_state` action, representing the communication of the new state to the parent. Likewise, the `send_command` actions and `receive_command` actions are synchronised, resulting in a `comm_command` action. Communication is enforced, and state and phase changes through the actions `move_state` and `move_phase` are also allowed. A slightly simplified example is shown in Translation 4.

```
allow({comm_state, comm_command,
       move_state, move_phase, ignored_command},
  comm({receive_state|send_state -> comm_state,
        receive_command|send_command -> comm_command},
    ...
 ||
    X_CLASS(configuration(2, 1, [3],
                      (lambda k:CLASS. [])[Y_CLASS -> [3]]),
                    S_OFF, (lambda i:ID. S_OFF), [],
                    WhenPhase, [], 0)
 ||
    Y_CLASS(configuration(3, 2, [4, 5],
                       (lambda k:CLASS. [])[Z_CLASS -> [4, 5]]),
                     S_OFF, (lambda i:ID. S_OFF), [],
                     WhenPhase, [], 0)
));
```

Translation 4: Simplified instantiation of the tree hierarchy. This describes, among others, an FSM of class X with a child of class Y, that in turn has children of class Z. Note that the parent of the X class FSM and the children of the Y class FSM are not shown in this instantiation.

Our automatic translation generates the instantiation from the database that stores all information on the hierarchy.

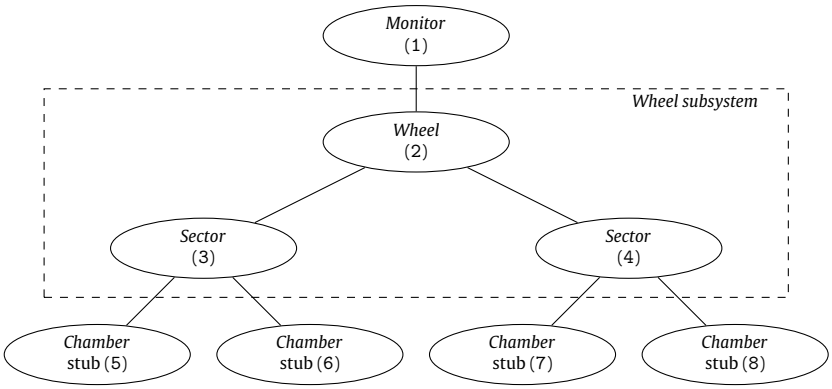

Fig. 3. A schematic overview of our model of the *Wheel* subsystem, and its used FSMs. The identifiers of the processes representing the FSMs are given between parentheses; these were used in our analyses.

4.2. Validating the formalisation of SML

The challenge in formalising SML is in correctly interpreting its language constructs. We combined two strategies for assessing and improving the correctness of our semantics: informal discussions with the development team of the language, supported by simulations run using the PVSS-II environment used for developing FSMs, and applying formal analysis techniques on sample FSMs taken from the control software.

The discussions with the SML development team were used to solidify our initial understanding of SML and its main constructs. Based on these discussions, we manually translated several FSMs into mCRL2, and validated the resulting processes manually using the available simulation and visualisation tools of mCRL2. This revealed a few minor issues with our understanding of the semantics of SML, alongside many issues that could be traced back to sloppiness in applying the translation from SML to mCRL2 manually.

In response to the latter problem, we eliminated the need for manually translating FSMs to mCRL2. To this end, we utilised the ASF+SDF meta-environment (see [5,36]) to rapidly prototype an automatic translator that, ultimately, came to implement the translation scheme we described in the previous subsection. The *Syntax Definition Formalism* (SDF) was used to describe the syntax of both SML and mCRL2, whereas the *Algebraic Specification Formalism* (ASF) was used to express the term rewrite rules that are needed to do the actual translation. Apart from the gains in speed and the consistency in applying the transformations that were brought about by the automation, the automation also served the purpose of formalising the informal semantics of SML, since the language is mapped to a language that does have a formal semantics.

The final details of our semantics were tested by analysing relatively well-understood subsystems of the control software in mCRL2. We briefly discuss our findings using a partly simplified subsystem, colloquially known as the *Wheel*, see Fig. 3. The Wheel subsystem is a component of the Resistive Plate Chamber (RPC) subdetector of the CMS experiment. It belongs to the barrel region of the RPC subdetector. Each Wheel subsystem contains 12 sectors, each sector is equipped with 4 muon stations which are made of Drift Tube chambers. We forego a detailed formal discussion of this subsystem (for details, we refer to [37]), but only address our analysis of this subsystem using formal analysis techniques, and the impact this had on our understanding of the semantics and the transformation. It is important to keep in mind that the analysis was conducted primarily to assess the quality of our translation; the correctness of the subsystem being only secondary.

The mCRL2 specification of the *Wheel* subsystem was obtained by combining the mCRL2 processes obtained by running our prototype implementation on each involved FSM. Generating the state space of the *Wheel* subsystem takes roughly one minute using the symbolic state space generation tools offered by the LTSmin tools [38]. This toolset can be integrated in the mCRL2 toolset. For the discussed configuration, the state space is still of modest proportions, measuring slightly less than 5 million states and 24 million transitions. Varying the number of children of class *Sector* causes a dramatic growth of the state space. Using 3 instead of 2 children of class *Sector* yields roughly 800 million states; using 4 children of class *Sector*, leads to 120 billion states, and generating the state space requires half a day.

Apart from repeating the simulations and visualisations, at this stage we also applied *model checking* to systematically probe the translation. Together with the development team of the *Wheel* subsystem, a few basic requirements were formalised in the modal μ-calculus [32], see Table 1.

The studied subsystem was considered to satisfy all stated properties. While smoothing out details in the translation of SML to mCRL2, the deadlock-freedom property was violated every now and then, indicating issues with our interpretation of SML. These were mostly concerned with the semantics of the blocking and non-blocking constructs of SML, and the complex constructs used to model the message passing between FSMs and their children.

Requirement two, *i.e.*, the absence of intermediate states in the *when phase* was violated only once in our verification efforts. A more detailed scrutiny of the run revealed a problem in our translation, which was subsequently fixed. This second requirement is in fact a strengthening of the more natural requirement that we study in the next section, requiring that an FSM cannot stay in the *when phase* indefinitely. Such a property ensures that the logic of an FSM eventually stabilises. Clearly,

Table 1
Basic requirements for the *Wheel* subsystem; `i:Id` denotes an identifier of an FSM; `i_c:Id` denotes a child of FSM `i`; `c:Command` denotes a command;`c2s(c)` denotes the state with the homonymous command name, *e.g.*, `c2s(ON) = ON`.

All properties are invariant properties; this is formalised through `nu X . [true] X && f`, which means that in every reachable state, property `f` should hold. The universal quantifiers in the formulae express that the invariants should hold for all combinations of nodes and commands.

1. Absence of deadlock, *i.e.* in every reachable state, a transition is enabled:

```
nu X. [true]X && <true>true
```

2. Absence of intermediate states in the *when phase*, *i.e.* in every reachable state, after every `move_state` action (expressed by the first `[move_state(i,s)]`), no `move_state` action is possible (expressed by `[exists s:State. move_state(i,s)]false`), until a `move_phase(i,ActionPhase)` action has occurred (`[!move_phase(i, ActionPhase)]Y`). Note that this implies stability of the *when phase*.

```
nu X. [true]X && forall i:ID.
       [exists s:State. move_state(i,s)](nu Y.
           [(!move_phase(i,ActionPhase))]Y
        && [exists s:State. move_state(i,s)]false)
```

3. Responsiveness, *i.e.* in every reachable state, the sending of a command from node `i` to `i_c` (denoted by `[comm_command(i,i_c,c)]`) is followed by a `comm_state(i_c, i, c2s(c))` action in a finite number of steps (denoted by `mu Y. [!comm_state(i_c, i, c2s(c))]`), furthermore, `<true>true` ensures that the least fixpoint subformula does not hold in deadlocked states.

```
nu X. [true]X && forall i,i_c:ID, c:Command.
       [comm_command(i,i_c,c)](mu Y.
           <true>true && [!comm_state(i_c,i,c2s(c))]Y)
```

4. Progress, *i.e.* in every reachable state, it must be possible to perform a `move_state` action (denoted by `<exists s:State. move_state(i,s)>true`) within a finite number of steps, which, like before, is denoted by the least fixpoint subformula.

```
nu X. [true]X && forall i:ID.
       mu Y. <exists s:State. move_state(i,s)>true ||
           (<true>true && [true]Y)
```

the latter property is satisfied if a `move_state` action of an FSM is always followed by a `move_phase` action preceding another `move_state` action, which is expressed by requirement two.

The third requirement, stating the inevitability of a state change by a child once such a state change has been commissioned, failed to hold. The violation is caused by the overriding of commands by subsequent commands that are issued immediately. Discussions with the development teams revealed that the violations are real, *i.e.*, they are within the range of real behaviour, suggesting that our formalisation of SML was adequate, but the requirement was too strict. The property was modified to ignore the spurious runs, resulting in the following property:

```
nu X. [true]X && forall i,i_c:ID, c:Command.
       [comm_command(i,i_c,c)](mu Y. <true>true &&
           [!(comm_state(i_c,i,c2s(c)) ||
              exists c':Command. comm_command(i,i_c,c'))]Y)
```

The fourth requirement also failed to hold. The formula that is checked requires that in all reachable states of the system, each FSM eventually moves to another state. The violation is similar spirited to the violation of the third requirement, and, again found to comply to reality. The weakened requirement that was subsequently agreed upon expresses the attainability of some state change:

```
nu X. [true]X && forall i:ID.
       mu Y. <exists s:State. move_state(i,s)>true || <true>Y
```

Neither visual inspection of the state space using 2D and 3D visualisation tools, nor simulation using the mCRL2 simulators revealed any further incongruences in our final formalisation of SML, sketched in the previous section.

5. Dedicated tooling for verification

Some desired properties, such as the absence of intermediate states, and, more generally, the absence of loops through the *when phase*, can be checked by analysing an FSM in isolation, using the transformation to mCRL2. However, the verification using the modal μ-calculus currently requires too much overhead to serve as a basis for lightweight tooling that can be integrated in the SML development environment, since their verification in mCRL2 requires their complete state space. We explored the possibilities of using *Bounded Model Checking* (BMC) [39,40] in an attempt to improve on this situation. In the rest of this section we focus on two problems that can be solved efficiently.

The first problem we consider is detecting *livelocks* that manifest themselves through loops in the *when phase* (*i.e.*, generalising the stability property expressed by requirement 2 in the previous section). We use an overapproximation for detecting such loops. This means that we are guaranteed to find all loops a system may exhibit, and the absence of loops

proves the absence of a particular type of livelock in the system. However, we may detect loops that cannot occur in practice because the circumstances under which these happen may not be feasible. In practice, the class of loops that is detected is considered relevant by the developers, and detection can be done in seconds. Even though not every loop we detect might occur in practice, it does indicate a design flaw, indicating unintended behaviour in the implementation.

The second problem we consider is the *mutual reachability* problem (*i.e.*, generalising requirement 3 of the preceding section). Basically, we use an underapproximation to detect whether there are states in an FSM that can never reach all other states in the FSM. If our analysis reveals there are indeed such states, it indicates that the FSM may become stuck in such states. While this may constitute desired behaviour, it can also indicate a design flaw. Note that, since we use an underapproximation, we may miss reachability issues that occur in practice.

We report on the results we obtained using the Python implementation of our dedicated verification tools, improving upon the performance of our prototype in ASF+SDF.

5.1. Loop detection

The basic idea of BMC is to check for a counterexample in bounded runs. If no bugs are found using the current bound, then the bound is increased until either a bug is found, the problem becomes intractable, or some pre-determined upper bound is reached upon which the verification is complete. The BMC problem can be efficiently reduced to a propositional satisfiability problem, and can therefore be solved by SAT methods. SAT procedures do not necessarily suffer from the space explosion problem, and a modern SAT solver can handle formulae with hundreds of thousands of variables or more, see *e.g.* [40].

We have applied BMC techniques for the detection of `move_to` loops. As an example of a `move_to` loop, consider the excerpt of the `TkControlGroup` FSM class in Listing 3, in which our tool found property violations. If an instance of `TkControlGroup` has one child of class `TkPowerGroup` in state `ANALOG_ON_RED` and one child of class `FwCaenChannelCtrl` in state `ON`, it will loop indefinitely between these two states. Once this happens, an entire subsystem may enter a livelock and become temporarily or permanently unresponsive; it may even crash and bring down the entire control system.

```
state: ANALOG_ON_RED
 ...
 when ( $ANY$TkPowerGroup not_in_state DIGITAL_ON_RED ) move_to LVMIXED
 ...

state: LVMIXED
 ...
 when ( $ANY$TkPowerGroup in_state {ON, HVMIXED} ) move_to HVMIXED
 when ( $ALL$FwCaenChannelCtrl in_state ON  and
        $ALL$TkPowerGroup in_state ANALOG_ON_RED ) move_to ANALOG_ON_RED
 ...
```

Listing 3: An excerpt from the `TkControlGroup` FSM that exhibits a loop within the *when phase*.

This specific error was caused by an overlooked copy and paste from the *when clauses* of another state. The error can easily be fixed by replacing the condition to exit from `ANALOG_ON_RED` with `when ( $ANY$TkPowerGroup not_in_state ANALOG_ON_RED ) move_to LVMIXED`. With this fix incorporated, the FSM does not exhibit any loops. We convert the loop detection problem into a graph problem as follows. Let $\mathcal{F}$ be an FSM and $\mathcal{M}$ be a Kripke structure. A state in $\mathcal{M}$ corresponds to the combined state of $\mathcal{F}$ and its children, *e.g.*, if $\mathcal{F}$ is in state `ON` and has two children which are in state `OFF`, then the corresponding state in $\mathcal{M}$ is (`ON`, `OFF`, `OFF`). There is a transition between two states s_1 and s_2 in $\mathcal{M}$ if and only if s_1 can do a `move_to` action to s_2 in $\mathcal{F}$. Moreover, every state in $\mathcal{M}$ is an initial state. It thus suffices to inspect $\mathcal{M}$ instead of $\mathcal{F}$, as stated by the following lemma:

Lemma 1. *$\mathcal{F}$ contains a loop of `move_to` actions if and only if $\mathcal{M}$ contains a loop.*

We next translate the problem of detecting a loop in $\mathcal{M}$ into a SAT problem. First, we consider executions of length k; afterwards, we show that we can statically choose k such that we can find every loop.

Let the predicate *in_state* be defined as follows: *in_state*(s, p, i) holds if and only if the process with identifier p is in state s after i steps. We assign the identifier zero to the FSM under consideration and the numbers 1, 2, 3, ... to its children. The resulting formula will have three components: the *state constraints*, the *transition relation* and the *loop condition*.

Using the state constraints, we ensure that each FSM is always in exactly one state. Moreover, the states of the children should not change during the execution of the *when phase*, per the semantics in the previous section. This is straightforwardly expressed as a Boolean formula on the *in_state* predicate.

Next, we encode the transition relation: the relation between *in_state*$(s, 0, i)$ and *in_state*$(s', 0, i+1)$ for every i. In other words: the `move_to` steps the parent process is allowed to take. This involves converting the *when clauses* for each state of the parent FSM, taking care the semantics as outlined in the previous section is reflected. The last ingredient is the loop

condition: if *in_state*$(s, 0, 0)$ holds, then *in_state*$(s, 0, i)$ must hold for some $i > 1$, indicating that the parent returned to the state in which it started.

The final SAT formula is obtained by taking the conjunction of the state constraints, the transition relation and the loop condition. It is not hard to see that if this formula is satisfiable, then there is a loop in $\mathcal{M}$ and hence in $\mathcal{F}$. It is more difficult to show that if there is a loop, then the formula is satisfiable. We need an intermediary lemma before we can prove this result.

Lemma 2. *Let $\mathcal{F}$ be an FSM. Suppose that $\mathcal{F}$ has children c_1 and c_2 of class t, both of which are in state s. Then removing c_2 will not affect the decision that $\mathcal{F}$ takes in the when phase.*

Proof. Let $\mathcal{F}$' be a copy of $\mathcal{F}$ without c_2. We prove this lemma by induction on SML expressions, showing that no SML expression can distinguish between the configuration before and after the removal. We have to consider the following three categories:

- *Simple*: expressions of the form `$(ANY|ALL)$P (in_state|not_in_state) X`, *e.g.*, `$ANY$P in_state A`.
- *Compound*: an expression that is a conjunction, disjunction or negation of expressions, *e.g.*, `($ALL$P in_state A) and (not ($ALL$P' in_state B))`.
- *Multistate*: a term of the form `$(ANY|ALL)$P (in_state|not_in_state) {X1,...,XN}`, *e.g.*, `$ANY$P in_state {A, B}`.

Note that a multistate (sub-)expression can always be rewritten to a compound expression of simple expressions. If no simple expression can distinguish between $\mathcal{F}$ and $\mathcal{F}$', then no compound expression can distinguish between $\mathcal{F}$ and $\mathcal{F}$' either. Hence, it remains to show that no simple expression can distinguish between $\mathcal{F}$ and $\mathcal{F}$'. Let s' be a state other than s. It is easy to verify that the four simple expressions of the form `$(ANY|ALL)$P (in_state|not_in_state) s` evaluate to the same result in $\mathcal{F}$ and $\mathcal{F}$' and that the four simple expressions of the form `$(ANY|ALL)$P (in_state|not_in_state) s'` evaluate to the same result in $\mathcal{F}$ and $\mathcal{F}$'. This covers all cases, completing the proof. □

Let n be the total number of states of the FSM $\mathcal{F}$ and let n_t be the total number of states of each child class t. We now prove that we can find all loops in $\mathcal{F}$.

Theorem 3. *All possible loops in $\mathcal{F}$ can be found by considering paths of length at most n in the fixed FSM configuration C, consisting of n_t children of each child class t.*

Proof. Since $\mathcal{F}$ has n states, a longest possible loop contains at most n states. Since every state in $\mathcal{M}$ is an initial state, every possible loop can be found by doing n steps from an initial state.

It remains to show that all loops can be found by considering configuration C. Let C' be the configuration obtained by adding a nonzero amount of children of class t to C for each child class t. Assume that $\mathcal{F}$ admits a loop $\ell = s_1, \ldots, s_k, s_1$ for C' but not for C. By the pigeon hole principle, C' must have at least two children c_1 and c_2 of some class t^*, both in some state s^*. By Lemma 2, we can remove c_2 from C' to obtain a configuration C'' which again admits loop ℓ. Repeating this exhaustively, we obtain a configuration equal to C that admits loop ℓ. Contradiction. This completes the proof. □

As a consequence of the above theorem, we can find all possible local loops that are present in an FSM in isolation.

In practice, the number of children of a particular class t is typically well below n_t, meaning that some of the loops that can be detected using n_t children of class t may not be present in the actual system. While such configurations may appear in future versions of the control system, signalling the presence of a loop in configurations that do not occur in practice ultimately will result in engineers ignoring the outcome of the analysis of our tools. However, using the FSM database described in Section 2.1, the information about the *actual* configurations of the control system can be retrieved. The same technique as outlined above can be applied to these configurations.

5.2. The FSM state reachability problem

A second desirable behavioural property of an FSM is that all states should remain potentially reachable during the execution of an FSM. That is, the FSM should not become trapped in a subset of its states. While we can again easily encode this property into the modal μ-calculus, we use a more direct approach to detect violations of this property by constructing a graph that captures all potential state changes. For this, we determine whether there is a configuration of children such that $\mathcal{F}$ can execute a `move_to` action from a state s to a state s'; again, of course, following the semantics as outlined in the previous section. Doing so for all pairs (s, s') of states of $\mathcal{F}$ yields a graph encoding all possible state changes of $\mathcal{F}$. Note that also for this reachability property, we can build a more accurate graph using the information found in the FSM database by determining the set of potentially enabled edges using configurations that actually occur in production.

Computing the strongly connected components (SCCs) of the thus obtained graph gives sufficient information to pinpoint violations to the reachability property: the presence of more than a single SCC means that one cannot move back and forth these SCCs (by definition of an SCC), and, therefore, their states. Note that this is an under-approximation of all errors that can potentially exist, as the actual reachability dynamically depends on the configuration of the children of an FSM. Still, as the state change graph of the `HCAL_DCS_Node` FSM class in Fig. 4 illustrates, issues can be found in production FSMs:

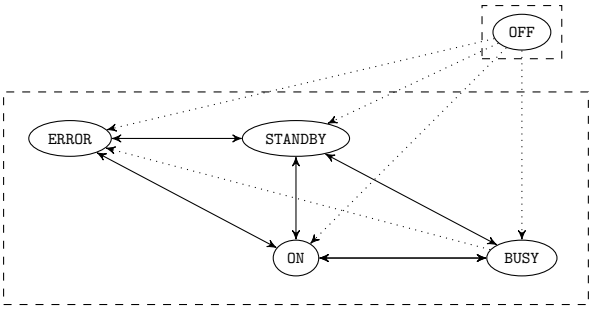

Fig. 4. The state change graph for the HCAL_DCS_Node FSM class. The solid lines are bidirectional; the dotted lines are unidirectional state changes. The SCCs are indicated by the dashed frames.

Table 2
Results of our BMC loop detection tool.

	Month	#Configurations	#Nodes with local loops	Time (s)
CMS	11-2011	408	1,695 of 8,326	64
	01-2012	568	1,659 of 9,038	81
	03-2012	564	1,546 of 9,045	82
	05-2012	578	1,302 of 9,064	79
LHCb	11-2011	1106	58 of 16,139	126
ATLAS	05-2012	1746	107 of 12,963	271
ALICE	05-2012	410	712 of 4,623	96

the OFF state can never be reached from any of the other states. Even though each state in the HCAL_DCS_Node class has a clause that specifies a transition to its OFF state, our tool found particular configurations in the control system for which these transitions would never become enabled. Using the graphs generated by our tools, such issues are quickly explained and located.

5.3. Results

The results using our dedicated tools for performing these behavioural sanity checks on isolated FSMs are very satisfactory. The checks were performed on the FSM databases of all LHC experiments, *viz.* CMS, LHCb, ATLAS, and ALICE,[1] using a 2006 laptop with an Intel T2400 Core 2 Duo processor and 1.5GB of RAM, *i.e.* modest requirements by today's standards. Note that similar experiments with the ASF+SDF prototype take several hours.

The results for local loop detection are reported in Table 2. For CMS four different snapshots of the control software are available, whereas for the other experiments, we have only got a single snapshot. The time of the snapshot is reported in the "Month" column. The number of configurations that must be checked is reported in the third column. Since the same configuration can occur multiple times in the hierarchy, detection of a loop in a single configuration can give rise to multiple nodes with the same loop; this is reported in the fourth column. The time it takes to detect loops in all configurations is reported in seconds in the final column.

Note that the only nodes that can exhibit loops are those that have children, hence the leafs of the hierarchy are excluded from the total number of nodes reported in the fourth column. For CMS, November 2011, there are 408 different configurations. Approximately 8300 of the 27,500 nodes have children, and roughly 1700 of these 8300 nodes have the potential to loop, which totals to 20% of the nodes that could contain local loops.

A manual inspection of a logfile of the operational control system revealed that one loop manifested itself at least three times on several nodes on a single day. An excerpt of the logfile containing the loop is shown in Fig. 5. No occurrence of the other detected loops was found in the logfiles of that month.

Interestingly, most of the loops we detected involved two states only. Note that the size of the average FSM class, in general more than 100 lines of SML code and at least two children, means that even short loops such as the ones identified so far remain unnoticed and are hard to pinpoint.

Apart from local loops, we also verified pairwise reachability in all configurations of the control systems. These results are reported in Table 3. Columns in this table are similar to the ones in Table 2, except that the time is reported in minutes instead of seconds.

We again elaborate further on the results for CMS, November 2011. The 408 configurations that were verified expand to 8326 nodes in the hierarchy. Of these nodes, 837 were shown to contain reachability problems, which amount to over 10%

[1] We present partial results of ALICE here.

```
...
Sun Nov 06 15:23:57 2011 - [PIXELBARREL_BMI_S7] in state [ANALOG_ON_RED]
Sun Nov 06 15:23:57 2011 - [PIXELBARREL_BMI_S7] in state [LVMIXED]
Sun Nov 06 15:23:57 2011 - [PIXELBARREL_BMI_S7] in state [ANALOG_ON_RED]
Sun Nov 06 15:23:57 2011 - [PIXELBARREL_BMI_S7] in state [LVMIXED]
Sun Nov 06 15:23:57 2011 - [PIXELBARREL_BMI_S7] in state [ANALOG_ON_RED]
Sun Nov 06 15:23:57 2011 - [PIXELBARREL_BMI_S7] in state [LVMIXED]
Sun Nov 06 15:23:57 2011 - [PIXELBARREL_BMI_S7] in state [ANALOG_ON_RED]
...
Sun Nov 06 15:38:08 2011 - [PIXELBARREL_BMI_S7] in state [ANALOG_ON_RED]
Sun Nov 06 15:38:08 2011 - [PIXELBARREL_BMI_S7] in state [LVMIXED]
Sun Nov 06 15:38:08 2011 - [PIXELBARREL_BMI_S7] in state [ANALOG_ON_RED]
Sun Nov 06 15:38:08 2011 - [PIXELBARREL_BMI_S7] in state [LVMIXED]
Sun Nov 06 15:38:08 2011 - [PIXELBARREL_BMI_S7] in state [ANALOG_ON_RED]
Sun Nov 06 15:38:08 2011 - [PIXELBARREL_BMI_S7] in state [LVMIXED]
...
```

Fig. 5. Logfile with evidence of the loop detected by our BMC loop detection tool.

Table 3
Results of our BMC reachability detection tool.

	Month	#Configurations	#Nodes with reachability violations	Time (min.)
CMS	11-2011	408	837 of 8,326	16
	01-2012	568	1,353 of 9,038	19
	03-2012	564	1,313 of 9,045	19
	05-2012	578	903 of 9,064	18
LHCb	11-2011	1106	430 of 16,139	25
ATLAS	05-2012	1746	2,089 of 12,963	61
ALICE	05-2012	410	1,836 of 4,623	23

of the non-leaf nodes in the hierarchy. The increase in reachability violations in CMS between November 2011 and January 2012 is caused by an extended maintenance period, in which our tools were not yet available to the FSM developers.

Using our Python implementation, using a standard SMT solver such as Yices [41] as backend, and running the analysis on off-the-shelf hardware, local loops can be detected in about a minute, and reachability violations can be detected within the hour for the complete control software of a detector. Running checks on configurations in isolation is even quicker, which means that the checks can easily be incorporated in the design cycle of the FSMs. The checks have been integrated in the development environment used at CERN. For a description of the integration, and an exploration of additional checks that can be implemented as a dedicated tool, see [42].

6. Conclusion

We discussed and studied the State Machine Language (SML) that is used for programming the control software of the CMS experiment and several other large experiments running at the Large Hadron Collider. To fully understand the language, we formalised it using the process algebraic language mCRL2. The quality of our formalisation was assessed using a combination of simulation and visualisation of the behaviour of FSMs in isolation and formally verifying small subsystems using model checking. To facilitate, among others, the assessment, the translation of SML to mCRL2 was implemented using the ASF+SDF meta-environment. Based on our understanding of the semantics of SML, we have built dedicated tools for performing sanity checks on isolated FSMs. Using these tools we found several issues in the control system. These tools have been well-received by the engineers at CERN, and have recently been integrated in the development environment.

Our formalisation of SML opens up the possibility of verifying realistically large subsystems of the control system; clearly, it will be one of the most challenging verification problems currently available. In our analysis of the *Wheel* subsystem, we have only used a modest set of tools for manipulating the state space. Attempts to use compositional verification based on behavioural equivalence reductions were not successful. Symmetry reduction, partial order reduction, parallel exploration techniques, and abstract interpretation were not considered at this point. It remains to be investigated how such techniques fare on this problem.

Acknowledgements

We thank Giel Oerlemans, Dennis Schunselaar and Frank Staals from the Eindhoven University of Technology for their contribution to a first draft of the ASF+SDF translation. We also thank Frank Glege, Robert Gomez-Reino Garrido, Lorenzo Masetti and Giovanni Polese from the CERN CMS DAQ group, Stefan Schlenker from the ATLAS experiment and Ombretta Pinazza from the ALICE experiment for their support and advice, and Clara Gaspar from the LHCb experiment and Boda Franek for discussions on SML. Jaco van de Pol is thanked for his help with the LTSmin toolset. First author's work has been supported in part by a Marie Curie Initial Training Network Fellowship of the European Community's Seventh framework

programme under contract number (PITN-GA-2008-211801-ACEOLE). The fifth author's research has been partially funded by the Netherlands Organisation for Scientific Research (NWO) under grant number 612.000.937

Appendix. Supplementary data

Supplementary data associated with this article can be found, in the online version, at doi:10.1016/j.scico.2012.11.009.

References

[1] O. Holme, M. González-Berges, P. Golonka, S. Schmeling, The JCOP Framework, Technical Report CERN-OPEN-2005-027, CERN, Geneva, 2005.
[2] B. Franek, C. Gaspar, SMI++ object-oriented framework for designing and implementing distributed control systems, IEEE Transactions on Nuclear Science 52 (2005) 891–895.
[3] C. Gaspar, B. Franek, SMI++—Object-oriented framework for designing control systems for HEP experiments, Computer Physics Communications 110 (1998) 87–90.
[4] J.F. Groote, A.H.J. Mathijssen, M.A. Reniers, Y.S. Usenko, M.J. van Weerdenburg, Analysis of distributed systems with mCRL2, in: Process Algebra for Parallel and Distributed Processing, Chapman Hall, 2009, pp. 99–128.
[5] M. van den Brand, A. van Deursen, J. Heering, H. de Jong, M. de Jonge, T. Kuipers, P. Klint, L. Moonen, P. Olivier, J. Scheerder, J. Vinju, E. Visser, J. Visser, The ASF+SDF meta-environment: a component-based language development environment, in: R. Wilhelm (Ed.), Proc. of Compiler Construction, in: LNCS, vol. 2027, Springer, 2001, pp. 365–370.
[6] E.M. Clarke, E.A. Emerson, Design and synthesis of synchronization skeletons using branching-time temporal logic, in: Logic of Programs, Workshop, Springer-Verlag, London, UK, 1982, pp. 52–71.
[7] J.-P. Queille, J. Sifakis, Specification and verification of concurrent systems in CESAR, in: Proceedings of the 5th Colloquium on International Symposium on Programming, Springer-Verlag, London, UK, 1982, pp. 337–351.
[8] E.M. Clarke, E.A. Emerson, A.P. Sistla, Automatic verification of finite-state concurrent systems using temporal logic specifications, ACM Transactions on Programming Languages and Systems 8 (1986) 244–263.
[9] Formal Systems (Europe) Ltd, FDR2 user manual, 2010.
[10] G.J. Holzmann, The SPIN Model Checker — Primer and Reference Manual, Addison-Wesley, 2004.
[11] J. Burch, E. Clarke, K. McMillan, D. Dill, L. Hwang, Symbolic model checking: 10^{20} states and beyond, in: Proceedings of Logic in Computer Science, LICS, 1990, pp. 428–439.
[12] E. Clarke, R. Enders, T. Filkorn, S. Jha, Exploiting symmetry in temporal logic model checking, Formal Methods in System Design 9 (1996) 77–104.
[13] E.A. Emerson, A.P. Sistla, Symmetry and model checking, Formal Methods in System Design 9 (1996) 105–131.
[14] M. Abadi, L. Lamport, Conjoining specifications, ACM Transactions on Programming Languages and Systems 17 (1995) 507–535.
[15] H.R. Andersen, Partial model checking, in: Proceedings of the 10th Annual IEEE Symposium on Logic in Computer Science, LICS'95, IEEE Computer Society, Washington, DC, USA, 1995.
[16] A.J. Mooij, N. Goga, Dealing with non-local choice in IEEE 1073.2s standard for remote control, in: D. Amyot, A.W. Williams (Eds.), System Analysis and Modeling, in: LNCS, vol. 3319, Springer, 2005, pp. 257–270.
[17] A.J. Mooij, N. Goga, W. Wesselink, D. Bosnacki, An analysis of medical device communication standard IEEE 1073.2, in: Proc. 2nd IASTED Int. Conf. on Communication Systems and Networks, ACTA Press, 2003.
[18] M. Devillers, D. Griffioen, J. Romijn, F. Vaandrager, Verification of a leader election protocol: formal methods applied to IEEE 1394, Formal Methods in System Design 16 (2000) 307–320.
[19] J. Romijn, Analysing industrial protocols with formal methods, Ph.D. Thesis, 1999.
[20] I. van Langevelde, J. Romijn, N. Goga, Founding firewire bridges through promela prototyping, in: Proc. 17th Int. Symposium on Parallel and Distributed Processing, IPDPS'03, IEEE Computer Society, Washington, DC, USA, 2003, p. 239.
[21] M. Fruth, Probabilistic model checking of contention resolution in the IEEE 802.15.4 low-rate wireless personal area network protocol, in: Proc. ISOLA'06.
[22] M. Duflot, L. Fribourg, T. Herault, R. Lassaigne, F. Magniette, S. Messika, S. Peyronnet, C. Picaronny, Probabilistic model checking of the CSMA/CD protocol using PRISM and APMC, in: Proc. AVoCS 2004, ENTCS, vol. 128, pp. 195–214.
[23] A. Mathijssen, A. Pretorius, Verified design of an automated parking garage, in: L. Brim, B. Haverkort, M. Leucker, J. van de Pol (Eds.), Formal Methods: Applications and Technology, in: LNCS, vol. 4346, Springer, 2007, pp. 165–180.
[24] J.F. Groote, J. Pang, A.G. Wouters, Analysis of a distributed system for lifting trucks, JLAP 55 (2003) 21–56.
[25] J.J.A. Keiren, M.D. Klabbers, Modelling and verifying IEEE Std 11073-20601 session setup using mCRL2, in: G. Lüttgen, S. Merz (Eds.), Proc. AVoCS 2012, in: Electronic Communications of the EASST, vol. 53 (in press).
[26] D. Remenska, T.A.C. Willemse, K. Verstoep, W. Fokkink, J. Templon, H.E. Bal, Using model checking to analyze the system behavior of the lhc production grid, in: CCGRID, pp. 335–343.
[27] F.P.M. Stappers, M.A. Reniers, Verification of safety requirements for program code using data abstraction, in: ECEASST 23, 2009.
[28] C. Gaspar, J. Schwarz, Controlling a large physics experiment; a communication issue, Control Engineering Practice 4 (1996) 257–264.
[29] J. Baeten, T. Basten, M. Reniers, Process Algebra: Equational Theories of Communicating Processes, in: Cambridge Tracts in Theoretical Computer Science, vol. 50, Cambridge University Press, 2010.
[30] D. Kozen, Results on the propositional mu-calculus, Theoretical Computer Science 27 (1983) 333–354.
[31] R. Mateescu, Vérification des propriétés temporelles des programmes parallèles, Ph.D. Thesis, Institut National Polytechnique de Grenoble, 1998.
[32] J.F. Groote, T.A.C. Willemse, Model-checking processes with data, Science of Computer Programming 56 (2005) 251–273.
[33] S. Cranen, J.F. Groote, M.A. Reniers, A linear translation from CTL* to the first-order modal μ-calculus, Theoretical Computer Science 412 (2011) 3129–3139.
[34] J. Bradfield, C. Stirling, Modal logics and mu-calculi, in: Handbook of Process Algebra, Elsevier, North-Holland, 2001, pp. 293–332.
[35] J. Groote, J. Keiren, A. Mathijssen, B. Ploeger, F. Stappers, C. Tankink, Y. Usenko, M. van Weerdenburg, W. Wesselink, T. Willemse, J. van der Wulp, The mCRL2 toolset, in: WASDeTT 2008.
[36] P. Klint, A meta-environment for generating programming environments, ACM Transactions on Software Engineering and Methodology 2 (1993) 176–201.
[37] P. Paolucci, G. Polese, The detector control systems for the CMS resistive plate chamber, 2008. CERN-CMS-NOTE-2008-036, see http://cdsweb.cern.ch/record/1167905.
[38] S.C.C. Blom, J.C. van de Pol, M. Weber, LTSmin: distributed and symbolic reachability, in: T. Touili, B. Cook, P. Jackson (Eds.), in: LNCS, vol. 6174, Springer, 2010, pp. 354–359.
[39] A. Biere, A. Cimatti, E.M. Clarke, Y. Zhu, Symbolic model checking without BDDs, in: Proc. of TACAS, in: LNCS, vol. 1579, Springer, 1999, pp. 193–207.
[40] A. Biere, A. Cimatti, E.M. Clarke, O. Strichman, Y. Zhu, Bounded Model Checking, Advances in Computers 58 (2003) 118–149.
[41] B. Dutertre, L.M. de Moura, A fast linear-arithmetic solver for DPLL(T), in: T. Ball, R.B. Jones (Eds.), CAV, in: Lecture Notes in Computer Science, vol. 4144, Springer, 2006, pp. 81–94.
[42] S.J.J. Leemans, Validation of CERN's finite state machines, Master's Thesis, Eindhoven University of Technology, 2012.