Electronic Communications of the EASST
Volume 35 (2010)

Proceedings of the 10th International Workshop on Automated Verification of Critical Systems (AVoCS 2010)

Automated Support for the Design and Validation of Fault Tolerant Parameterized Systems: a case study

F. Alberti, S. Ghilardi, E. Pagani, S. Ranise, and G.P. Rossi

14 pages

Guest Editors: Jens Bendisposto, Michael Leuschel, Markus Roggenbach
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

Automated Support for the Design and Validation of Fault Tolerant Parameterized Systems: a case study

F. Alberti[1], S. Ghilardi[2], E. Pagani[2], S. Ranise[1]*, and G.P. Rossi[2]

FBK-Irst, Trento, Italia[1]
Università degli Studi di Milano, Milano, Italia[2]

Abstract: We propose a methodology to use the infinite state model checker MCMT, based on Satisfiability Modulo Theory techniques, for assisting in the design of fault tolerant algorithms. To prove the practical viability of our methodology, we apply it to formally check the agreement property of the reliable broadcast protocols of Chandra and Toueg.

Keywords: Fault Tolerant Algorithms, Infinite state model checking, Parameterized verification, Satisfiability Modulo Theories

1 Introduction

Algorithms for ensuring fault tolerance are key ingredients in many applications such as avionics, networking, transportation, and industrial plants. There is an increasing demand to integrate (formal) validation in the design process of these algorithms as they are often part of safety critical systems. When validation fails, the designer will benefit from tracking the sequence of events that led to an incorrect state to recover the error. To productively integrate formal verification in the design phase, tools should be able to return such error traces. Automating verification for fault tolerant algorithms turns out to be a daunting task as they are often parametric (i.e. they are designed to work with an arbitrary finite number of processes) so that checking that an algorithm satisfies a certain property requires to prove it regardless of the number of processes.

In this paper, we propose the use of an infinite state model checker for safety properties, called MODEL CHECKER MODULO THEORIES (MCMT) [4], to assist in the design of this class of algorithms. MCMT is particularly suitable for this purpose because it is based on a declarative framework in which parametric algorithms can be naturally specified, and uses the Satisfiability Modulo Theories (SMT) technology to automate their verification. Such solvers have already proved to be quite effective for solving SMT problems resulting from the encoding of large verification problems as witnessed by the SMT-LIB initiative (`http://www.SMT-LIB.org`). The **first contribution** of this paper is the definition of a sub-set of the formal framework [3] underlying MCMT, which is suitable for the specification of distributed fault tolerant algorithms (first part of Section 2). This contribution is less obvious than it may appear at first glance. In fact, while several available specification languages can express the same class of algorithms that we consider in this paper (see, e.g., [12]), the available analysis techniques for these languages

* Partially supported by the "Automated Security Analysis of Identity and Access Management Systems (SIAM)" project funded by Provincia Autonoma di Trento in the context of the "Team 2009 - Incoming" COFUND action of the European Commission (FP7).

are not completely automated and require substantial user intervention. Worst of all, the user should be an expert not only of the systems being verified but also of the automated techniques employed for the analysis. Instead, one of the goals of this work is to find the right balance between expressiveness and the possibility to design a highly automated analysis technique that requires only high-level user interventions, of the kind an expert in the field of fault tolerant algorithms can provide. We believe this paper makes a step in this direction.

Fault tolerant algorithms are assumed to work in an environment where certain types of failures may happen while others are ensured not to occur. For example, the simplest of such models is the crash failure where processes may halt at any time. A slightly more realistic model consists of considering the possibility that a process may omit a message as well as crashing at any time. One of the most interesting features of MCMT is to natively support the stopping failure model [6]. The **second contribution** of the paper is a technique to transform the set of transitions so as to consider more complex failure models (second part of Section 2). The underlying idea is to add a local state variable to each process as a flag signaling if the process is faulty or not and to refine transitions specification according to faulty/non-faulty behaviors.

Then, we propose a design methodology for parametrised and fault tolerant algorithms that exploits the previous two contributions. Roughly, the design phases consider the specification of the algorithm and of its safety property, the choice of the failure model in which the protocol should operate (e.g., crash-failure or send-omission), and finally the validation through the model checker. This schema can be easily blended with a standard incremental and iterative approach to design. For example, the user can perform partial verifications during early development phases, change its failure model into a more realistic one, determine invariant properties of the system he is building, and so on, thereby getting various kinds of benefits from the automated support. The **third contribution** of this paper is to apply the methodology identified above to replay the design of the reliable broadcast algorithms of Chandra and Toueg in [14] (Section 3). The results of MCMT confirm those of the pen-and-paper proof in the published paper showing the practical viability of our techniques.

Related work

The specification of a parametric system $\mathscr{S}$ is usually given in terms of the cardinality n of a parameter set (e.g., the set of nodes in a mutual exclusion protocol) and it is denoted with $\mathscr{S}(n)$. Its verification has a long history and it can be roughly classified as one of three types. The first approach (see, e.g., [2]) tries to find a cut-off value $\overline{n}$ for n so that the instance $\mathscr{S}(\overline{n})$ exposes all the possible behaviors which are relevant for the verification of certain properties. The drawback is that the cut-off value $\overline{n}$ is usually high and its verification turns out to be a daunting task.

The second approach is pioneered in [1]. The key idea is to symbolically represent sets of states with constraints and use them to explore the infinite state space of $\mathscr{S}(n)$, where n is an arbitrary natural number. In this method, the most important test is to understand when all the search space has been explored. The drawback is that the test depends on the topology of the system so that a change in the topology requires its re-implementation.

The third approach (somehow complementary to the second above) relies on *predicate abstraction* [7]. The idea is to abstract $\mathscr{S}(n)$ to a finite state system, to perform model checking on it, and then to refine spurious traces (if any) by using decision procedures or SMT solvers.

Table 1: Experimental results on some "standard" benchmarks

	Default setting				Best setting				
Problem	d	#n	#SMT	time	d	#n	#SMT	#i	time
Lamport	23	913	47574	120.62	23	248	19254	7	32.84
RickAgr	13	458	35355	187.04	13	458	35355	0	187.04
Szymanski_at	23	1745	424630	540.19	9	22	2987	42	1.25
German07	26	2442	121388	145.68	26	2442	121388	0	145.68
GermanBug	16	1631	41497	49.70	16	1631	41497	0	49.70

Legenda: 'd,' depth of the tree visited during backward search; '#n,' number of nodes in the explored search space; '#SMT,' number of invocations to the SMT solver, which is Yices (http://yices.csl.sri.com) in the current version of the system; '#i,' number of invariants synthesized by MCMT (this is listed only for the 'Best setting' because invariant synthesis is disabled by default); 'time,' total amount of time (in seconds) taken by the tool to solve the problem on a Pentium Intel 1.73 GHz with 1 GB Sdram running Linux Gentoo.

This technique has been implemented in several tools and is often combined with interpolation algorithms for the refinement phase. The main problem with predicate abstraction (as already pointed out in [8]) is that it must be carefully adapted when (universal) quantification is used to specify the transitions of the system or its properties, as is the case for the verification problems considered in this paper.

The approach underlying MCMT[1] and used in this paper is different in several respects to the three approaches listed above. It does not attempt to find a cut-off value but rather uses particular classes of first-order formulae to represent sets of states parametrised with respect to two classes of algebraic structures, one for data and one for the topology (see [3] for details). In this way, the basic operations underlying the exploration of the state space reduces to simple logical manipulations and to the solution of SMT problems containing universal quantifiers, which are both independent of the topology of the system (contrary to the second approach above). The current implementation of MCMT is orthogonal to the predicate abstraction approach since it is the state space of the original system that is considered while a primitive form of predicate abstraction (with no refinement) is only used to synthesize invariants of the system that are then used to prune the search space when possible (see [5] for a full account of the technique). These features enable MCMT to efficiently solve verification problems which are considered as benchmarks in the literature. To illustrate, we give a brief overview of the capabilities of MCMT on some selected problems concerning parametrised and distributed systems available in its distribution: 'Lamport,' 'Rick(art-)Agr(avala),' and 'Szymanski_at(omic)' are parametrised and distributed protocols for mutual exclusion (all correct), 'German07' and 'GermanBug' are two versions (the former is correct while the latter is bugged) of a cache-coherence protocol (we point the reader to the tool web-page for details). In Table 1, we report the performances of the tool with two settings: 'Default' is when MCMT is invoked without any option and 'Best Setting' is when the tool is run with non default options. In Section 3, we report the performances of MCMT on significantly more difficult problems than those in Table 1 (compare the columns

[1] Available on-line at http://www.dsi.unimi.it/~ghilardi/mcmt

marked with '#n' in Table above and the row '# nodes' in Table 2 to get an idea of the size of the explored state space), thereby showing its scalability.

Finally, we remark that although in this paper we consider only the verification of safety properties, our framework supports also the parametric verification of a sub-class of liveness properties (see again [3] for details). The implementation of this technique is left to future work.

2 Fault tolerant algorithms in MCMT

Several fault tolerant algorithms are described as systems executing a series of "rounds" (see, e.g., [10]). A round has two phases: in the first, each process sends a message to some or all of the other processes (messages will depend on the current state of the sending process); in the second phase, each process changes its state according to its current state and the collection of messages it received in the first phase. Messages are transferred instantaneously from senders to recipients between the two phases. The processes operate in lockstep: all of them perform the two phases of the current round, then move on to the first phase of the next round, and so on. Each process may fail independently for a variety of reasons, that are classified according to a certain "failure model."

In the following, we briefly describe how to formalize this class of distributed systems and their safety (i.e. properties of the kind "nothing bad can happen") in the infinite state model-checker MCMT. We focus on a *small* sub-set of the abstract syntax and semantics of the tool, which is sufficient to our needs. The interested reader is pointed to [3] for a full account of the underlying formal framework and to [4] (and the user manual, available on-line from the web-page of MCMT) for a description of the concrete syntax and the options of the tool.

Let $\mathscr{S}$ be a distributed system. In MCMT, the states of $\mathscr{S}$ are described by a finite tuple $\underline{a} = a_1, \ldots, a_s$ of *array variables*, which maps indices to data. A *safety problem* for $\mathscr{S}$ is specified by (i) a formula I of first-order logic characterizing the set of *initial states* of $\mathscr{S}$; (ii) a finite set $Tr = \{\tau_1, \ldots, \tau_m\}$ of first-order formulae characterizing the *transitions* of $\mathscr{S}$; and (iii) a first-order formula U describing the set of *unsafe states* (usually obtained by complementing the safety property we would like $\mathscr{S}$ to satisfy). The verification of a safety property is reduced to repeatedly computing the pre-image of U (i.e. the set of states from which U can be reached by taking a transition from Tr, denoted with $Pre(\tau, U)$) until a fix-point is reached (*fix-point check*) or the intersection of the current set of backward reachable states with the set of initial states is found to be non-empty (*safety check*). If a fix-point is reached and the intersection with I is empty, then the system is *safe* with respect to U; otherwise, it is *unsafe*. In the second case, MCMT also returns an *error trace*, i.e. a sequence of transitions leading the system from an initial to an unsafe state; thereby, MCMT can be seen as both a verifier and a debugger.

The repeated computation of pre-images and the fix-point and safety checks are known as *backward reachability procedure* [1]. In MCMT, this procedure is organized so as to visit a tree whose nodes are labelled by the formulae of the pre-images of U with respect to the transitions in Tr. More precisely, the procedure is as follows. First, the root node is created and labelled by U. Then, the pre-image of U w.r.t. a certain $\tau \in Tr$ is computed and a son of the root is created and labelled with $Pre(\tau, U)$. Then, this process is recursively performed by considering the new node as the root of a sub-tree. Indeed, the visit of the tree is interleaved with fix-point

and safety checks so as to decide when the visit can be stopped. To mechanize this, MCMT puts some *constraints on the format* of I, Tr, and U so that (a) the class of formulae describing the set of backward reachable states are closed under pre-image computation and (b) both fix-point and safety checks can be reduced to decidable logical problems, called Satisfiability Modulo Theories (SMT) problems, for a certain class of first-order formulae (containing universal quantifiers). Sufficient conditions to ensure (a) and (b) are precisely identified in [3] and their effect is to restrict the class of algebraic structures that can be used in the specification of the indexes (and the elements) in the arrays. Since (universal) quantifiers are needed to encode safety and fix-point checks arising from the verification of practically relevant classes of systems (such as the fault tolerant algorithms considered in this paper), the classes of structures identified above allow us to show the decidability of the resulting SMT problems by integrating a quantifier instantiation procedure with "standard" SMT solving techniques for quantifier-free formulae.

Before formally describing the format of I, U, and Tr used in this paper, we informally characterize it as corresponding to a sub-set of the logic underlying UCLID [9]. UCLID logic contains uninterpreted functions (to represent arrays), λ-expressions (to model updates of the arrays), equality and (linear) ordering (to specify the guards of the updates). The two arithmetic operators used in UCLID (namely, successor and predecessor functions) are not used here. As in UCLID, we allow for the situation where data stored in an array can be used to dereference the same array or others. Because of the use of this last feature (which turns out to be very useful for the specification of the algorithms in [14]), the termination of the backward reachability procedure is not guaranteed because the sufficient conditions for termination identified in [3] are not satisfied. However, on the algorithms in [14], we were able to have termination as discussed in Section 3. We now formally describe the format of I, U, and Tr that allows us to naturally specify parametric fault tolerant algorithms.

Specifying fault tolerant algorithms. Array variables are typed, i.e. indices and data must have a type. Indices represent the identifiers of processes and their type is a fixed (but unknown) *finite subset* of the natural numbers, denoted by `INDEX`. In this way, the topology of the system is that of a set of processes that can be distinguished via their identifiers, which can be compared w.r.t. equality. If more complex topologies are needed (e.g., identifiers must be sorted according to an order because one of the processes must be elected as coordinator according to a certain ascending order), MCMT supports the specification of richer index types, such as linear orders, graphs, and forests. Data can take values over a wide range of types, such as integers, Booleans, process identifiers, and program counters. In fact, MCMT accepts all these types and also user-defined ones. For the goals of this work and to simplify the exposition, in the following, we assume that the type of data is always that of integers (denoted with $\mathbb{Z}$). If a is an array variable and i is an index variable, then $a[i]$ denotes the value of the *local* variable a of process i. It is possible to encode a *shared* variable b among the processes by considering b as an array variable but requiring that $\forall i, j.(b[i] = b[j])$.[2]

[2] The MCMT specification language (even if restricted as in this paper) is quite general and expressive. As such, it can cover very heterogeneous problems. The examples here and below refer *just to the formalization of our case studies from Section 3*; we hope they can help the reader to get acquainted with our formal language. The MCMT files with the complete specifications of all algorithms considered in this paper can be downloaded at http://www.falberti.it/reliableBroadcast.

Example 1 *The specification of fault tolerant algorithms includes a bounded number of rounds of executions before the algorithm terminates. Hence, if, e.g., the algorithm comprises only three rounds, then we can introduce in* MCMT *the following array variable* `round`*, mapping indices to integers storing the value of the current round of execution of the fault tolerant algorithm. The variable* `round` *is shared so that we also assume that* $\forall i,j.(\texttt{round}[i] = \texttt{round}[j])$*. Furthermore, since the number of rounds is bounded to 3, we can also assume that* $\forall i.(1 \leq \texttt{round}[i] \wedge \texttt{round}[i] \leq 3)$*.* MCMT *accepts the specification of this kind of constraints on array variables.*

Before continuing, we need to introduce the following notion. A *constraint* is a conjunction of formulae of the kind $t \bowtie u$, where $\bowtie \in \{>,\geq,<,\leq,=,\neq\}$ and t,u are either (i) integer constants (such as $0,3,7$) or (ii) expressions of the form i_l or $a_k[i_l]$, where a_k is an array variable among the state variables in $\underline{a} = a_1,\ldots,a_s$ and i_l is an index variable. We reserve the letters $C,D,\ldots$ for constraints and the notation $C(\underline{i})$ means that at most the index variables in $\underline{i}$ may occur in C.

The formula specifying the set of initial states is of the form $In(\underline{a}) := \forall i.C(i)$, where C is a constraint, usually containing expressions of the form $a_k[i] = v$ for v a constant value and a_k an array variable among the state variables in $\underline{a} = a_1,\ldots,a_s$; C might constrain a subset of the state variables in $\underline{a}$ to assume certain values, while leaving unspecified the values of the others.

Example 2 *At the beginning of the algorithm, no process of the system has yet decided on the value of the message being broadcast (*$\texttt{state}[i] = \mathit{false}$*), no one is or has been the coordinator (*$\texttt{coord}[i] = \mathit{false}$*,* $\texttt{aCoord}[i] = \mathit{false}$*), no message request has been sent by any process (*$\texttt{request}[i] = \mathit{false}$*) and the system is at the beginning of the first round (*$\texttt{round}[i] = 1$*,* $\texttt{done}[i] = \mathit{false}$*). This can be formalized as follows:*

$$In \;:=\; \forall i.\left(\begin{array}{lllllll} \texttt{round}[i] = 1 & \wedge & \texttt{state}[i] = \mathit{false} & \wedge & \texttt{coord}[i] = \mathit{false} & \wedge \\ \texttt{aCoord}[i] = \mathit{false} & \wedge & \texttt{done}[i] = \mathit{false} & \wedge & \texttt{request}[i] = \mathit{false} & \end{array}\right).$$

The formula specifying the set of unsafe states is of the form $U(\underline{a}) := \exists i_1 \cdots \exists i_n.\, C(i_1,\ldots,i_n)$.

Example 3 *The key safety property for our algorithms is agreement expressing the fact that if any two distinct processes have decided on the value of the message being broadcast, then the value is the same. The complement of this property is represented by the following formula*

$$U(\underline{a}) \;:=\; \exists i_1,i_2.(i_1 \neq i_2 \wedge \left(\begin{array}{c} \texttt{state}[i_1] = \mathit{true} \wedge \texttt{state}[i_2] = \mathit{true} \wedge \\ \texttt{decisionValue}[i_1] \neq \texttt{decisionValue}[i_2] \end{array}\right)),$$

which characterizes all the states whose array variables `state` *and* `decisionValue` *contain* at least *two distinct indices* i_1 *and* i_2 *such that the values of* `state` *at the two indices are true and the values of* `decisionValue` *at the two indices are distinct. Notice the use of the word "at least" in the previous sentence which suggests the idea that formulae in the above format represent all those states (potentially infinitely many) that contain at least the processes identified by the existentially quantified variables and that satisfy the constraints in the conjunction following the quantifier.*

The formula specifying a transition $\tau(\underline{a},\underline{a}') \in Tr$ (as usual, a primed state variable indicates

the value of the variable after the execution of the transition) may have one of the two forms[3]

$$\exists i_1.(C(i_1) \wedge \forall \ell.D(i_1,\ell) \wedge \bigwedge_{k=1}^{s} a'_k = \lambda j.Upd_k(j,i_1)), \quad (1)$$

$$\exists i_1,i_2.(C(i_1,i_2) \wedge \forall \ell.D(i_1,i_2,\ell) \wedge \bigwedge_{k=1}^{s} a'_k = \lambda j.Upd_k(j,i_1,i_2)), \quad (2)$$

where the updates $Upd_k(j,i_1), Upd_k(j,i_1,i_2)$ are *functions defined by cases*

$$F(\underline{i},j) := \texttt{case of} \quad \{C_1(\underline{i},j) : t_1; \cdots C_r(\underline{i},j) : t_r\};$$

where the constraints $C_1,\ldots,C_k$ are exhaustive and mutually exclusive[4] and $t_1,\ldots,t_r$ are numerical constants or expressions of the forms $i_l, j, a_k[i_l], a_k[j]$ (in other words, the function $\lambda j.F(\underline{i},j)$ maps j to the value t_1 if C_1 holds, the value t_2 if C_2 holds, etc.). The intuitive reading of (1) and (2) is the following. For a transition to fire, a guard must be satisfied. The guard consists of a local condition C and a global condition D, i.e. a process i_1 must awake (satisfying condition C) or two distinct processes i_1,i_2 must synchronize and all (other) processes must satisfy condition D (notice that one or both components can be tautological, because the empty conjunction *true* is a constraint). When the guard is satisfied, the transition fires and the array variables $\underline{a} = a_1,\ldots,a_s$ are updated according to the case-defined functions $Upd_1,\ldots,Upd_s$.

Example 4 *For a process, going from one round (say the first) to the next (the second) can be formalized as follows:*

$$\exists i_1. \left(\begin{array}{l} \texttt{round}[i_1] = 1 \wedge \texttt{request}[i_1] = true \wedge \texttt{coord}[i_1] = true \wedge \\ \forall \ell.(\texttt{done}[\ell] = true) \wedge \\ \texttt{round}' = \lambda j.\texttt{case of}\{(j = i_1) : 2; \quad (j \neq i_1) : 2\} \end{array} \right),$$

i.e. if the current round is one, process i_1 has already received a request, it is the coordinator, and all processes have finished the execution of the round, then the shared variable `round` *is set to two. Implicitly, the other state variables are not changed, i.e. $\alpha' = \alpha$ for each $\alpha \neq$* `round` *in $\underline{a}$ is added to the formula above. Notice that the update of* `round` *maintains the invariant $\forall i,j.(\texttt{round}[i] = \texttt{round}[j])$ introduced in Example 1.*

At this point, the reader may be skeptical about the usability of first-order logic as a specification language. The goal of the above discussion is only to present the formal framework for the specification of fault tolerant algorithms used by MCMT precisely and concisely, not to argue for its adoption as an appropriate specification language. Rather, we regard it as a target language for translators from higher level (and more natural) specification languages. In fact, we developed one of such a language to carry out the experience described in Section 3.

Failure Models and MCMT. Besides the fault tolerant algorithm itself, the second key ingredient in the specification of a fault tolerant system is the environment in which the algorithm is assumed to work. The environment should specify how and when the processes in the system

[3] Notice that at most two existentially quantified variables occur in (1) and (2). This is enough for the protocols in [14] and experience shows that they are sufficient for the specification of many classes of systems as witnessed by the example problems in the distribution of MCMT.

[4] I.e. the formulae $\neg(\bigvee_{i=1}^{r} C_i)$ and $C_j \wedge C_i$ (for $i \neq j$) are all unsatisfiable.

may fail: in [14], a taxonomy of failure models is proposed. In the following, we consider two such models. In the *crash failure model* (CFM), a process may fail by halting prematurely at any time; it behaves correctly until it halts and after that it does nothing, i.e.—in practice—it disappears from the protocol. CFM has been widely adopted in the literature under different names (e.g., it is called *stopping failure model* in [10]). In the *send omission failure model* (SOFM), a process may crash or may omit to send some of the messages it should send according to the algorithm. SOFM is more realistic than CFM in that it considers message losses because of network congestion or misbehavior. In both models, failure is ascribed to processes, that are considered *faulty* for the whole execution of the algorithm; non-faulty processes are called *correct*.

Example 5 *For reliable broadcast [14], there is a group G of processes, and a message m sent to the group by one of its members. The members of G decide about the delivery of m so as to satisfy the following safety property, called* agreement*: if a correct process delivers m, then all correct processes must also deliver m. Notice that agreement is unambiguously defined only when a failure model is chosen since 'correct' is the complement of 'faulty,' which, in turn, is defined by the failure model.*

We discuss how failure models are dealt with by MCMT. The tool implicitly adopts the CFM so that faulty processes are those that are crashed. In this model, all universally and existentially quantified variables occurring in the formulae In, U, and the transitions (1), (2) above are implicitly assumed to *range over correct* (i.e. non-crashed) *processes* (w.r.t. the CFM). This means that $\exists i.\varphi(i)$ must be read as "there exists a non-crashed process i such that φ" and that $\forall i.\varphi(i)$ as "for all non-crashed process i, it happens that φ," where φ is a formula satisfying the requirements explained above.

Example 6 *The correct reading of the formula for the agreement property of Example 3 is: 'there exists two distinct and non-crashed processes i_1 and i_2 such that the values of* `state` *at i_1 and i_2 are true and the values of* `decisionValue` *at i_1 and i_2 are distinct.'*

The fact that a process can fail at any time in the CFM is modelled in MCMT by automatically adding a suitable transition τ_{crash} to the set Tr of transitions specified by the user. Interestingly, the presence of τ_{crash} complicates the intuitive reading of formulae (1) and (2) for transitions. In fact, this makes it possible to fire a transition without the need to check the global condition D. This is possible by applying τ_{crash} to all those processes not satisfying D so that they are all faulty and the universally quantified variable in D does not range over them any more. For a formal account of the way MCMT handles the CFM, the reader is pointed to [6].

While MCMT provides native support for the CFM, the handling of the SOFM is more complex. In fact, it is necessary to add a new array state variable `faulty` mapping indices to Booleans such that `faulty`$[i]$ is false (true) when the process i is correct (faulty, resp.) w.r.t. the SOFM. Indeed, the new variable must be mentioned in the safety properties to be checked and explicitly updated in the set Tr of transitions specified by the user so as to model the situation when the process executing a transition is correct or faulty. Notice that crash failures will still be implicitly handled by MCMT as explained above; hence, we need to explicitly take into consideration only the send omission failures.

Example 7 *The agreement property in Example 3 should be written as follows when considering the SOFM:*

$$\exists i_1, i_2.(i_1 \neq i_2 \wedge \left(\begin{array}{c} \mathtt{state}[i_1] = \mathit{true} \wedge \mathtt{faulty}[i_1] = \mathit{false} \wedge \\ \mathtt{state}[i_2] = \mathit{true} \wedge \mathtt{faulty}[i_2] = \mathit{false} \wedge \\ \mathtt{decisionValue}[i_1] \neq \mathtt{decisionValue}[i_2] \end{array} \right)),$$

where it is explicitly required that the two processes i_1 and i_2 are correct (i.e. the flag `faulty` *is false at both i_1 and i_2).*

Let us consider the reliable broadcast algorithms of [14], where a transition in the first round models the situation where the undecided processes (i.e. $\mathtt{state}[i_1] = \mathit{false}$*) send a request to the coordinator:*

$$\exists i_1, i_2.(i_1 \neq i_2 \wedge \left(\begin{array}{l} \mathtt{round}[i_1] = 1 \wedge \mathtt{done}[i_1] = \mathit{false} \wedge \mathtt{state}[i_1] = \mathit{false} \wedge \\ \mathtt{coord}[i_2] = \mathit{true} \wedge \mathtt{request}' = \lambda j.\mathit{true} \wedge \\ \mathtt{done}' = \lambda j.\mathtt{case\ \ of}\{(j = i_1) : \mathit{true};\ (j \neq i_1) : \mathtt{done}[j]\} \end{array} \right)),$$

where the array variables are those introduced in Examples 1 and 2. This formula alone correctly captures the behavior of the transition only if we adopt the CFM. In fact, if we use the SOFM, the formula above only captures the case when the sending of the request to process i_2 (the coordinator) from a process i_1 is successful. To specify the situation when this is not the case, we must add the following transition:

$$\exists i_1, i_2.(i_1 \neq i_2 \wedge \left(\begin{array}{l} \mathtt{round}[i_1] = 1 \wedge \mathtt{done}[i_1] = \mathit{false} = \mathtt{state}[i_1] \wedge \mathtt{coord}[i_2] = \mathit{true} \wedge \\ \mathtt{faulty}' = \lambda j.\mathtt{case\ \ of}\{(j = i_1) : \mathit{true};\ (j \neq i_1) : \mathtt{faulty}[j]\} \wedge \\ \mathtt{done}' = \lambda j.\mathtt{case\ \ of}\{(j = i_1) : \mathit{true};\ (j \neq i_1) : \mathtt{done}[j]\} \end{array} \right)),$$

modelling that process i_1 is faulty since it fails to send the request to i_2.

It is possible to mechanize the task of adding transitions modelling the failures due to the omission of sending messages so that the designer is only required to specify the set of transitions for the correct processes. To give an intuition of how this is possible, it is sufficient to notice that we can add decorations to the transitions specifying which state variables are affected by the action of sending a message and who is the process sending it.

Integrating MCMT **in the design of fault tolerant algorithms**. We propose the following methodology for the design of parametric and fault tolerant algorithms (see also Figure 1). The designer first specifies the initial states, the transitions Tr, and the complement U of the safety property that the system should satisfy. Then, he selects one of the possible failure models (e.g., CFM or SOFM). Once U, I and Tr have been processed so as to take into consideration the chosen failure model, MCMT is invoked. If the tool returns 'safe', then the designer can change the failure model, experiment with variants and refinements of the algorithms, or consider a new safety property. Otherwise, if the tool returns 'unsafe', the designer can analyze the error trace so as to locate the problem in the transitions or in the property that has been submitted to the model checker. This methodology can be naturally integrated with iterative approaches to the design of algorithms.

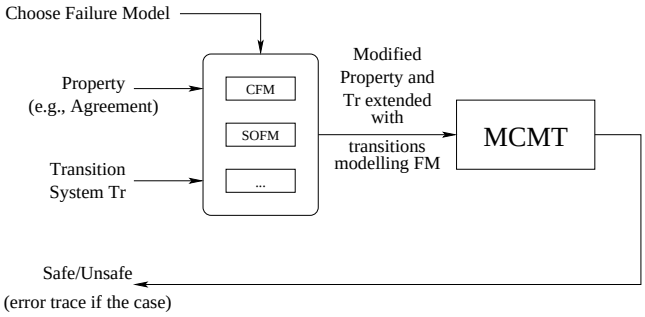


Figure 1: The methodology for designing fault tolerant algorithms

3 Case Study: Chandra and Toueg Algorithms

As a proof of concept of our proposed technique, we report our experience in applying it to the design and validation of the distributed algorithms proposed in [14] to solve the *reliable broadcast problem*. The safety property to be satisfied is the Agreement property hinted in Example 5. The algorithms are parametric in the number of processes participating in the protocol. In particular, we consider a group consisting of n processes communicating through reliable links of a fully connected network, where $n > 0$ is fixed but unknown.

In [14], the reliable broadcast problem is solved for the two failure models considered in Section 2 and a more general model, that we do not consider here. The algorithms in [14] are round-based – as described at the beginning of Section 2 – and they are presented incrementally. First, an algorithm for the simplest failure model is presented, and its correctness is proven. Then, the authors show misbehaviors that are possible when adopting the algorithm in the SOFM. Subsequently, a modified version of the algorithm is proposed – and its correctness proven – in the more complex model. No automated formal method is used, just informal reasoning in natural language.

As we have already observed at the end of Section 2, our methodology nicely blends with an iterative design approach followed by the authors of [14]. The starting point for the CFM in [14] is a simple algorithm called Algorithm 1 (see **Algorithm 1** *without the highlighted parts* for details). The algorithm has been formalized in a high-level specification language that is automatically translated to the input syntax of MCMT, which is the concrete counterpart of the framework described in Section 2. The formal specification consists of nearly 140 lines of text. Communication is represented by updating the process variables following the reception of a message. Besides the transitions formalizing some part of the pseudo-code of the algorithm, some transitions have been added so as to (i) allow progress of the system in case no requests are received in Round 1; (ii) describe the switching to a new coordinator either at the end of Round 4 for the current one, or when the current coordinator crashes; (iii) specify the behavior of already decided processes. The safety of the algorithm w.r.t. the agreement property (as formalized in Example 3, i.e. when CFM is assumed) is quickly established by MCMT (see first column, second line of Table 2). Once the correctness of the algorithm in the CFM has been established, the authors of [14] argue that the same algorithm is unsafe in the SOFM. To show this, they produce an error trace involving two coordinators, where a faulty (but not crashed) coordinator c_1 first

Algorithm 1 Pseudo-code for Algorithms 1, 1e, and 2

Initialization:
  if (p is the sender)
    then $estimate_p \leftarrow m$; $coord\_id_p \leftarrow 0$;
    else $estimate_p \leftarrow \perp$; $coord\_id_p \leftarrow -1$;
  $state_p \leftarrow undecided$;
End Initialization

for $c \leftarrow 1, 2, ..., f+1$ **do** // Process c becomes coordinator for four rounds
  Round 1:
    All *undecided* processes p send *request* $(estimate_p, coord\_id_p)$ to c;
    if (c does not receive any request) **then** it skips rounds 2 to 4;
      else $estimate_c \leftarrow estimate_p$ with largest $coord\_id_p$;
  Round 2:
    c multicasts $estimate_c$;
    All *undecided* processes p that receive $estimate_c$ do
      $estimate_p \leftarrow estimate_c$ **and** $coord\_id_p \leftarrow c$;
  Round 3:
    All *undecided* processes p that do not receive $estimate_c$ send(NACK) to c;
  Round 4:
    if (c does not receive any NACK) **then** c multicasts *Decide*; **else** c HALTS;
    All *undecided* processes p that receive *Decide* **do**
      $decision_p \leftarrow estimate_p$;
      $state_p \leftarrow$ DECIDED;
end for

omits to send his estimate $estimate_1$ to a process c_2 that will become the next coordinator c_2, and then sends a decide to at least one correct process p. At this point, c_2 becomes the coordinator and sends its estimate $estimate_2$ to an undecided correct process q; the following sending of a 'decide' message from c_2 to q brings the system in an inconsistent state. We tried to do the same following our methodology. First, we selected the SOFM and produced the new set of transitions, as discussed in Example 7. Second, the complement of the agreement property is modified so as to take into account the newly introduced state variable, again as discussed in Example 7. Finally, MCMT is invoked on the resulting safety problem and it quickly concludes the unsafety of the system with an error trace. For better readability, we have manually transformed the error trace found by the tool into the message sequence chart shown in Figure 2. The trace found by MCMT consists of eleven transitions and it involves only one coordinator instead of two as in [14]. Since each time a new coordinator is elected, the algorithm re-starts from round 1, our trace is shorter than the one illustrated in [14] where two processes played the role of coordinator.

By analyzing the error trace, the authors of [14] suggest to add a state variable `nack` (for *negative acknowledgment*), used in the additional Round 3 and the `if` statement in Round 4 (highlighted part in **Algorithm 1**). Clearly, processes may fail in sending `nacks`. In order to describe this new algorithm, that will be called in the following 'Algorithm 1e', six new

Table 2: MCMT performances

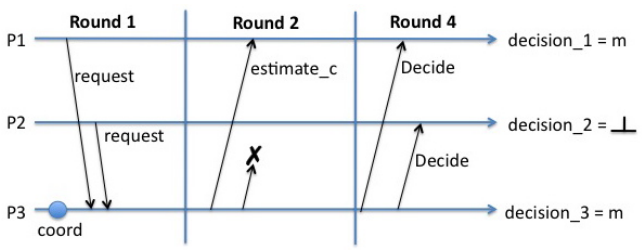
	Algo. 1, CFM	Algo. 1, SOFM	Algo. 1e, SOFM	Algo. 2, SOFM
Safe (agreement)	Yes	No	No	Yes
time (sec)	1.18	17.66	1,709.93	4,719.51
# state vars	8	9	11	15
# transitions	13	13+3	16+6	22+6
# nodes	113	464	9,679	11,158
# SMT calls	2,792	20,009	1,338,058	2,558,986
Length unsafe trace	×	11	33	×
# invariants	×	×	×	19 (+**7**)

Timings are obtained on an Intel Core Duo 2.66 GHz with 2 GB, running Debian Linux.

transitions are needed corresponding both to the behaviors of correct and faulty processes that need to send a negative acknowledgment. (In the formalization, besides `nack`, another state variable is added to keep track of the fact that a process has received the estimate from the coordinator; this is why the number of state variables are eleven in column 3, line 3 of Table 2.) As argued in the original paper, this refinement of the algorithm to comply with the SOFM is found to be unsafe by MCMT (although it takes a significantly longer time to discover this as can be seen by comparing the values at line two of Table 2). The error trace comprises 33 transitions and, after a manual analysis, it is possible to see that it corresponds to the execution described in [14] in natural language.

As a further step toward the design of a correct algorithm for the reliable broadcast problem, [14] proposed that processes send more information with their requests, namely, the *estimate* they currently hold and the identity of the coordinator from which it has been received. The current coordinator does not impose its own decision; rather, it adopts and circulates the *estimate* associated with the most recent coordinator (highlighted parts in Initialization, Rounds 1 and 2 of **Algorithm 1**). The formalization consists of 15 state variables and 28 transitions (six of which are introduced to model send omission failures), see last column of Table 2. The up-front verification of agreement for this new version of the algorithm is quite problematic: MCMT runs for many hours without finding a fix-point. Even the invariant synthesis capabilities of the

Figure 2: Error trace produced by MCMT for algorithm 1 in the SOFM



tool [5] do not help here, although they have been proved quite effective in pruning the search space for the verification of other systems, see [5]. However, the possibility to interact with the tool by proving simpler properties (that an expert in fault tolerant algorithms is able to identify) and then telling MCMT to exploit them for proving more difficult ones is the key to the result recorded in the last column of Table 2. The inclusion of invariants does not affect the validity of the final result of the verification process, since invariants are verified by the tool before being used for the validation of other properties. Seven invariants suggested by the designer have been included, that were quite simple properties that do not require any deep understanding of the algorithm (e.g., "there is only one coordinator at a time"). The use of these invariants (plus 19 more automatically found by the invariant synthesis techniques available in the tool) allowed us to validate the safety of Algorithm 2 in the SOFM. The tool also validated the three lemmas used in [14] to perform the pen-and-paper proof of the correctness of the algorithm.

4 Discussion

We have described a methodology that exploits the automated analysis capabilities of the model checker MCMT to support an incremental and iterative approach to designing fault tolerant algorithms. The methodology has been put to test on a group of significant algorithms for reliable broadcast considered in [14]. Our experiments confirmed the finding of the original paper and the support for incrementality and refinement during the design phases.

There are two main lines for future work. First, we would like to consider more general failure models (e.g., general omission) and finish the formal validation of the reliable broadcast algorithms in [14]. Second, we intend to investigate how to refine our models so as to take into account temporal constraints that would allow us to consider more realistic models of the algorithms along the lines of [13].

Bibliography

[1] Abdulla, P. A., Cerans, K., Jonsson, B., and Tsay, Y.-K. General decidability theorems for infinite-state systems. In *Proc. of LICS*, pages 313–321, 1996.

[2] Arons, T., Pnueli, A., Ruah, S., Xu, J., and Zuck, L.. Parameterized Verification with automatically computed inductive assertions. In *Proc. of CAV'01*, LNCS 2102, 221–234, 2001.

[3] Ghilardi, S., Nicolini, E., Ranise, S. and D. Zucchelli. Towards SMT Model-Checking of Array-based Systems. In *Proc. of IJCAR*, LNCS 5195, 2008.

[4] Ghilardi, S. and Ranise, S. MCMT: A Model Checker Modulo Theories. Accepted for publication in *Proc. of IJCAR*, 2010.

[5] Ghilardi, S. and Ranise, S. Goal Directed Invariant Synthesis for Model Checking Modulo Theories. In *TABLEAUX*, LNAI, pages 173–188. Springer, 2009.

[6] Ghilardi, S. and Ranise, S. A note on the stopping failure model. Unpublished note available at http://homes.dsi.unimi.it/~ghilardi/mcmt/stop_fail_note.pdf

[7] Graf, S. and Saïdi, H. Construction of abstract state graphs with PVS. In *Proc. of CAV 1997*, volume 1254 of *LNCS*. Springer, 1997.

[8] Lahiri, S. K. and Bryant, R. E. Predicate abstraction with indexed predicates. *ACM Transactions on Computational Logic (TOCL)*, 9(1), 2007.

[9] Lahiri, S.K., Seshia, S.A. and Bryant, R.E. Modeling and Verification of Out-of-order Microprocessors using UCLID. Proc. Intl. Conf. on Formal Methods in Computer-Aided Design (FMCAD), LNCS 2517, 143–159, 2002.

[10] Lynch, N. A.: Distributed Algorithms. *Morgan Kaufmann*, 1996.

[11] Manna, Z. and Pnueli, A. A hierarchy of temporal properties. In *Proc. of PODC '90*, pages 377–410. ACM Press, 1990.

[12] Manna, Z. and Pnueli, A. Temporal Verification of Reactive Systems: Safety. Springer-Verlag, 1995.

[13] Rushby, J.: Systematic Formal Verification for Fault-Tolerant Time-Triggered Algorithms. *Proc. 6th Working Conference on Dependable Computing for Critical Applications*, IEEE Computer Society Press, 203—222 (1997).

[14] Toueg, S., Chandra, T. D.: Time and Message Efficient Reliable Broadcast. *Proc. 4th Intl. Workshop on Distributed Algorithms*, LNCS 486, 289–303 (1990).