HAL
open science

A formal approach for the synthesis and implementation of fault-tolerant industrial embedded systems

Wei-Tsun Sun, Alain Girault, Gwenaël Delaval

To cite this version:

Wei-Tsun Sun, Alain Girault, Gwenaël Delaval. A formal approach for the synthesis and implementation of fault-tolerant industrial embedded systems. SIES'2015: 10th IEEE International Symposium on Industrial Embedded Systems, Jun 2015, Siegen, Germany. hal-01165686

HAL Id: hal-01165686

https://inria.hal.science/hal-01165686v1

Submitted on 22 Jun 2015

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Public Domain

A formal approach for the synthesis and implementation of fault-tolerant embedded systems

ABSTRACT

We demonstrate the feasibility of a complete workflow to synthesize and implement correct-by-construction fault tolerant distributed embedded systems consisting of real-time periodic tasks. Correct-by-construction is provided by the use of discrete controller synthesis (DCS), a formal method thanks to which we are able to guarantee that the synthesized controlled system satisfies the functionality of its tasks even in the presence of processor failures. For this step, our workflow uses the HEPTAGON domain specific language and the SIGALI DCS tool. The correct implementation of the resulting distributed system is a challenge, all the more since the controller itself must be tolerant to the processor failures. We achieve this step thanks to the libDGALS real-time library (1) to generate the glue code that will migrate the tasks upon processor failures, maintaining their internal state through migration, and (2) to make the synthesized controller itself fault-tolerant.

1. INTRODUCTION

1.1 Safety critical embedded systems

Embedded systems account for a major part of critical applications (space, aeronautics, nuclear...) as well as public domain applications (automotive, consumer electronics...). Their main features are: (1) Critical real-time: unmet timing constraints may involve a system failure leading to a disaster; (2) Constrained resources: they rely on limited computing power and memory because of weight and encumbrance, power consumption (autonomous vehicles or portable devices), radiation resistance (nuclear or space), or price constraints (consumer electronics); and (3) Distributed and heterogeneous architecture: they are often distributed to provide enough computing power and to keep computing sites close to the sensors and actuators.

1.2 The need for formal methods

An embedded system being intrinsically critical, it is essential to ensure that it is tolerant to processor failures. This can even motivate the distributed scenarios, where the loss of one computing site must not lead to the loss of the whole application. We advocate that formal methods allow us to design and implement systems with guarantees on their fault-tolerance. We use discrete controller synthesis (DCS), the advantages of which being that the correctness of the resulting system is enforced in an automatic way. In our context, the controller synthesized by DCS maintains the functionality of the system, whatever the faults under some failure hypothesis. We propose to designers a sound methodology and tool flow for modeling multi-task and multi-processor distributed systems (including its functionality in terms of periodic tasks, the processor model, and the failure model) and synthesizing automatically a correct-by-construction fault-tolerant distributed implementation.

The output of our tool flow is a fault-tolerant distributed system with dynamic reconfiguration that is guaranteed to be correct thanks to DCS. A system consists of a set of periodic tasks placed in a configuration onto a set of processors. Upon the occurrence of a processor failure, tasks must be placed anew in another configuration, by migrating some of them onto other processors, so that execution can proceed. These configurations of the system have to be controlled according to a fault-tolerance policy, enforced by the synthesized controller. The properties of the controller are specified in terms of contracts on the tasks' behaviors and of several criteria to be optimized, for example the load balancing between the processors.

1.3 Contributions

We present the following contributions: (1) the modeling of multi-task and multi-processor distributed systems with constraints to enforce fault-tolerance; (2) the design flow to incorporate multi-variable optimization on the system model; (3) the protection of the controller with spatial redundancy and the implementation of an election algorithm to prevent a single point of failure; and (4) an automatic approach to generate the system model, to compile, and to map the implementation on the target distributed memory platform.

2. BACKGROUNDS

2.1 Fault-tolerance

We assume the following failure hypothesis: only the processors can fail, with a fail-silent model. That is, a processor is either active and works fine, or faulty and does not produce any output[1]. To tolerate such failures, we are going to make use of the intrinsic hardware redundancy offered by the distributed architecture: i.e., we do not wish to add extra processors but to use only the existing ones. Our goal is to apply failure recovery techniques (check-pointing and rollback), such that whenever a processor fails, the tasks

[1]Fail silence can be implemented, for instance with dual-lock architecture.

that were active on it will be dynamically migrated (resuming execution from the last saved checkpoint) on some other non-faulty processor. The new state of the system reached after such a recovery is degraded in the sense that less processors are now available, but the functionality is maintained since all the tasks are still being executed.

2.2 Discrete controller synthesis

DCS emerged in the 80's [14], with foundations in language theory. Its principle is, given two languages $\mathcal{M}$ and $\mathcal{D}$, to find a third language $\mathcal{C}$ such that $\mathcal{M} \cap \mathcal{C} \subseteq \mathcal{D}$. Here, $\mathcal{M}$ is called the system model, $\mathcal{D}$ the desired system, $\mathcal{C}$ the controller, and $\mathcal{M} \cap \mathcal{C}$ the controlled system model (CSM). As illustrated in Figure 1(a), the system model has a set of input I, and a set of outputs O to the environment. The set of inputs is partitioned into a set of uncontrollable inputs I_u, coming from the environment, and a set of controllable inputs I_c, provided by the controller. To control the system, the controller is then synthesized to compute the inputs I_c, from the inputs I_u and the state of the system model S (Figure 1(b)).

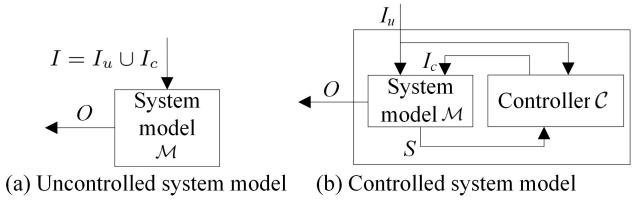


Figure 1: The use of discrete controller to obtain a controlled system model (CSM).

2.3 Property-enforcing layers

We use DCS within a property enforcing layer framework [1]. We start from a system model built as the parallel composition of a set of Labelled Transition Systems (LTS), which constitutes the initial uncontrolled system of Figure 1(a). Each LTS models one component of our multi-task multi-processor distributed system, e.g., a task, a processor, etc. We then add a list of constraints to specify the desired functionalities of the controlled system, in the form of assume/guarantee contracts in the domain specific language Heptagon [5]. Each single constraint is then given to the DCS tool Sigali [11], resulting in one property enforcing layer that enforces the given constraint on the controlled system. When reacting to the environment, the obtained controller sets the values of the controlled inputs I_c such that the controlled system satisfies the given constraints whatever the values of the uncontrollable inputs I_u.

The advantages of this method are twofold: on the one hand, the property enforcing layer is correct, because of the fact that it is the result of an exact and exhaustive computation. On the other hand, the automated nature of the process makes for an easy modifiability of designs, be it in the components behaviors or in the declarative properties; hence, a variety of global constraints can be experimented for a given system under study, providing an effective support in the design space exploration.

3. THE MOTIVATION SCENARIO

In this section, we specify our system model and failure hypothesis. We consider systems composed of: (1) a distributed heterogeneous architecture, consisting of a set of fail-silent processors and one stable memory, fully connected by reliable point-to-point communication links; and (2) a set of periodic tasks, with the possibility to run them on the different processors.

As a concrete example, we use a systems of 3 processors which will execute 3 tasks. The set of all processors is $P = \{\rho_1,\ \rho_2,\ \rho_3\}$, the of all tasks is $T = \{\tau_1,\ \tau_2,\ \tau_3\}$. Tasks run in a time-sharing manner, so that several tasks can be active on the same processor at the same time. Tasks can migrate from one processor to another when: (1) the processor where the task runs fails; and (2) when running a task on a processor violates some constraints of the system, e.g., exceeds the maximum load of the processor. Task migration is categorized as *strong migration*, that is, a task resumes its execution from the last checkpoint that was saved before the migration. This is implemented in the code of the task body with checkpointing and rollback (where to insert checkpoints is orthogonal to our problem and we assume it is done at periodic intervals). Checkpoints are saved to the single stable memory of the system.

The processors are embedded inside a fully connected network of point-to-point communication links. We assume that the communication links do not fail. Each task can be executed on any processor. The controller is a special task which is replicated on *all* processors to avoid having a single point of failure, but only one of these replica is active. The controller is in charge of sending control signals to all the tasks active in the system, for instance to trigger their migrations. Besides, each processor executes one heartbeat task that sends periodically an "alive" message to all the other processors, and one detector task that gathers all these messages. When a processor is detected to be faulty, because it did not send any "alive" message for a predefined duration[2], the controller steers the system to a new configuration (i.e., migrates the tasks that were running on this processor, but possibly also other tasks) to guarantee that all the tasks are running on healthy processors and to optimize criteria chosen by the designer (e.g., to balance the load). If the faulty processor was running the controller, then another processor is elected to activate its local replica of the controller. Election procedures are classic so we do not detail it here. When a faulty processor is repaired and comes back to life, only its heartbeat and detector tasks are activated, but the active controller will detect this and will decide what tasks need to be migrated on this repaired processor, for instance to optimize the overall processor load.

Each task is characterized by a set of criteria that can be, e.g., its work load, power consumption, quality of service, etc. The processors are heterogeneous, meaning that the characteristics of task executions can be different on each processor. For the example, we define criteria $Q1$ and $Q2$, as the weights for each task running on each of the processors, detailed in Table 1. We define $Q1^i_j$ and $Q2^i_j$ as the weights of $Q1$ and $Q2$ for a given task τ_i (where $\tau_i \in T$) running on a processor ρ_j (where $\rho_j \in P$). We assume the weights for each task running on each processor is constant. E.g., the weight of task 1 running on processor 3 is of $Q1^1_3 = 2$. Weights are additive: the total weight on the processor j is the sum of the weights of all the active tasks on this processor. This is appropriate since, here, Q1 models the processor load. For other kinds of criteria, other combination functions can be used (max, multiplication, etc). b_j is the quantitative bound on $Q1$ for processor ρ_j. E.g., b_1 can be the maximum

[2]Usually a duration equal to three heartbeat periods.

computation capacity for processor 1. Table 1 specifies that $b_1 = 5$, $b_2 = 4$, and $b_3 = 6$. In this example, the system will be controlled to ensure the following properties: (1) tasks can only execute on non-faulty processor; (2) $Q1$ on each processor does not excess the bound; (3) the sum of $Q1$ in the system is minimized; and (4) the sum of $Q2$ in the system is maximized.

	Criteria $Q1$			Criteria $Q2$		
	ρ_1	ρ_2	ρ_3	ρ_1	ρ_2	ρ_3
τ_1	4	4	2	3	5	3
τ_2	2	2	3	2	2	5
τ_3	2	3	4	2	2	5
Bound b_j	5	4	6			

Table 1: The characteristics of the task executions on the different processors.

4. THE PROPOSED APPROACH

In this section, the languages and tools used to create the controlled system model (CSM) and the run-time support are described. We first introduce the HEPTAGON [5] language which is used to model the system and to synthesize the discrete controller. libDGALS [15] is a software library implementing the Dynamic Globally Asynchronous and Locally Synchronous (DGALS) Model of Computation (MoC) to support task migration on the distributed memory target platforms.

4.1 The design flow

Figure 2 illustrates the design flow of implementing the motivation example. The design flow consists of two phases: (1) the modeling of the system and synthesizing the discrete controller, are shaded as dark grey and are detailed in Section 5, as described in [7], mainly in the HEPTAGON synchronous data-flow language; and (2) the implementation and the integration of the system, detailed in Section 6.

We first map the periodic tasks and the processors into LTSs (Figure 2(a)). The task model is the simplified representation of the task activities. The failure model represents the collective status of the system according to the processors' health (working or faulty). As Figure 2(b) illustrates, the system model $\mathcal{M}$ is composed together with the indicators of task migrations and the calculation of the criteria (Section 3). The controller $\mathcal{C}$ is then synthesized by providing the constraints to the system with the DCS tool (Figure 2(c)). The CSM is the composition of the system model and the synthesized controller (Figure 2(d)). The C code of the CSM is then generated by the HEPTAGON compiler and is integrated with the other components of the system programmed with libDGALS.

The dashed rectangles in Figure 2 are the *glue-logics* generated automatically by our framework for the system integration. For example, the CSM receives the alive messages from the heartbeat tasks, and the task status (e.g., task termination) from the support codes of the tasks (Figure 2(e)). The CSM is also equiped with the controller election and the replication logics to prevent single-point of failure (i.e., one of the CSM replica can take over from the failed CSM). The outputs of the CSM are connected to the tasks' LTSs, to coordinate task migration when necessary (Figure 2(f)).

4.2 Obtain the controlled system with HEPTAGON

HEPTAGON is a data-flow synchronous language extended with contracts, which express the properties to be enforced

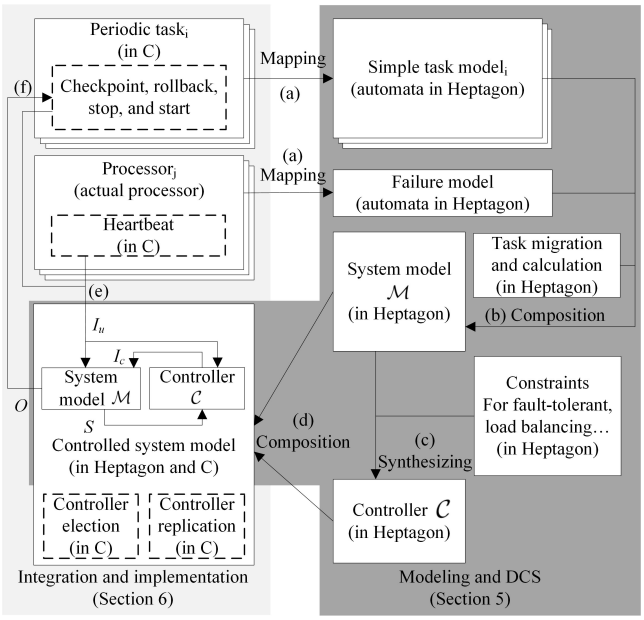


Figure 2: The proposed design flow.

on the resulting system, in our case fault-tolerance properties. Contracts are enforced by a controller computed by the SIGALI DCS tool [11]. An HEPTAGON program is compiled into a C file that implements its behavior, and into a Z3Z file that encodes the program into a symbolic transition system over the $\mathbb{Z}/3\mathbb{Z}$ domain [13]. This Z3Z file is then passed to SIGALI to generate a discrete controller enforcing, if possible, the contracts on the symbolic transition system. Because HEPTAGON does not provide programming constructs to optimize the system over the variable, we have implemented a wrapper to generate a Z3Z file so that the criteria (as shown, e.g., in Table 1) can be optimized automatically by SIGALI. The synthesized controller is produced as an HEPTAGON program, which is compiled into a C file. The synthesis fails if the contracts are impossible to enforce; this can occur if, e.g., not enough resources are provided or if the required bounds are too tight. Yet, the resulted controlled system cannot be executed as is.

The contribution of this paper is precisely to implement this controlled system onto a distributed memory architecture, such that it is indeed fault-tolerant. The challenge is threefold: first we must incorporate a failure detection mechanism (in the system abstract model used for DCS, failures are just discrete events), then we must incorporate a checkpointing mechanism to support the migration of the tasks (the controller only switches the system from one configuration to another one), and finally we must protect the controller itself from the possible failure of its processor. This is the role of the glue code and of the items in dashed boxes shown in Figure 2.

4.3 libDGALS: the library to program dynamic GALS systems

We choose libDGALS to be the run-time support for our approach for the following reasons: (1) libDGALS implements the DGALS model of computation (MoC), which is a superset of the synchronous MoC. On the other hand, since the synthesized controller is a synchronous HEPTAGON

program, it can be easily implemented in libDGALS and on the other hand the resulting distributed system is intrinsically asynchronous; and (2) libDGALS provides all the necessary programming constructs to implement dynamic systems with task migration over the distributed platforms.

In a nutshell, basic behaviors in DGALS systems are reactive and they interact with the environment continuously, hence they are called reactions. A reaction itself is a purely sequential execution unit (a function in C code). Concurrency is achieved by composing reactions with the synchronous product. A set of reactions that execute synchronously result a synchronous island called a Clock Domain (CD). Reactions in a CD communicate via internal signal broadcast as in Esterel [4]. Each CD runs at a different speed and reactions from different CDs communicate via channels. A channel is point to point, unidirectional, and uses CSP rendezvous [9] to guarantee data delivery between reactions.

The dynamicity of the DGALS systems comes from the creation of CDs at runtime (called activation). To allow communication between reactions of the newly created CD with the existing ones, channels need to be added at runtime. In our example, the synthesized CSM, the controller election mechanism, and controller replication are all integrated into the controller CD, which interacts with the task CDs and heartbeat CDs. The CDs are compiled and linked by the C compiler which are deployed to the DGALS programs over the distributed platform.

5. THE SYSTEM MODEL

The system is modeled with HEPTAGON with components named nodes. The system model consists of nodes of the following: (1) three task models, (2) one processors model, (3) criteria calculation, (4) one migration indicator, and finally (5) the top level node where the contracts are defined.

5.1 The task model

Each task τ_i is formally modeled by the LTS of Figure 3(a), drawn assuming that the task can be executed on the three processors of the considered architecture. It features an initial idle state I^i, a ready state R^i after reception of the request signal r^i, a terminal state T^i, and several active states A^i_j, representing task configurations, one for each processor in the system. Because there are three processors in the distributed platform, therefore each task LTS has three active states. In the state A^i_j, task τ_i is executed on processor ρ_j, until the occurrence of the event t^i. A transition caused by activation signal, a^i_j, from one active state to another active state, represents the migration of the task from one processor to another. For example, if τ_1 is running on ρ_1 (i.e., in the A^1_1 state) and a^1_2 is issued, the task will migrate to ρ_2 (i.e., enter the A^1_2 state). A migration could be decided as a reaction to a processor failure. But it could also serve to balance the load between several active processors, or to comply with the bound of $Q1$ of a processor. In terms of controller synthesis, the signals r^i and t^i will be uncontrollable (i.e., $\in I_u$), while the signals a^i_j will be controllable (i.e., $\in I_c$).

5.2 The processors and the system failure model

We assume that only the processors can fail with a fail-silent model. A processor which fails will stop sending heartbeats to the other processors. A processor can be restarted with no task executing on it. The restarted processor will resume sending the heartbeats to indicate its presence.

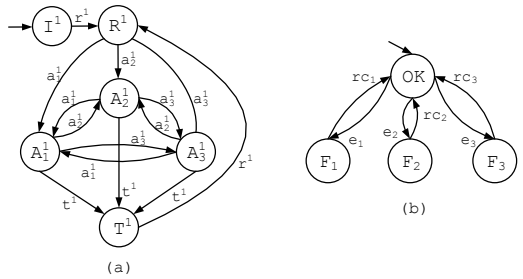


Figure 3: (a) Model of task τ_1 running on $P = \{\rho_1,\ \rho_2,\ \rho_3\}$; (b) Failure model with only one processor failure at a time and processor recovery.

The failure/recovery of ρ_j is captured by the input events e_j/rc_j. All the e_j are uncontrollable (i.e., $\in I_u$), to reflect the fact that a failure can occur at any time. It is therefore possible that all e_j could occur, meaning that all processors could fail. Of course, this would result in a total failure of the system, with no possibility at all to ensure the fault-tolerance of the system. No one expects a system to tolerate a failure of all the processors it is made of. Therefore, we need to specify the way the failures do occur, i.e., the number of processor failures allowed in the patterns that we consider.

We have chosen the processor failure model allowing one failure and single event transition as shown in Figure 3(b). By convention, OK is the initial state where all processor are healthy, while F_j denotes the failure of ρ_j. The failure model automaton outputs internal signals f_j to the other HEPTAGON nodes. In this sense, the failure model acts as a filter for the incontrollable events e_j. By convention, in any state F_j, we have $f_j = true$. Using different failure model allows the designer to explore different options.

5.3 The criteria and the migration indicators

To synthesize a controller that prevents the criteria $Q1$ from exceeding the bounds, as well the optimization on both $Q1$ and $Q2$, we present the criteria calculation node. It computes the sum of $Q1$ for each processor, according to the status of the tasks (the $task_i_on_processor_j$ signals). For instance, if task 1 and task 2 are running on processor 1, the sum will be $Q1^1_1 + Q1^2_1$.

We distinguish task migration and task restart. A task migration involves storing the checkpoint, terminating the task, and rolling-back the task on the new processor. A task restart only rolls-back the state of the task. The condition s_i to issue the migration signal for task τ_i is as follows:

$$s_i = a^i_{j'} \wedge \texttt{pre}(task_i_on_processor_j) \wedge \texttt{pre}(\neg f_j);$$

In the above formula, $\texttt{pre}(x)$ denotes the previous value of x. This means the task was active on processor ρ_j and migrates to $\rho_{j'}$ without the previous failure of the processor ρ_j. Note that the signals a^i_j, $task_i_on_processor_j$, and f_j are from the controller, task node, and processor failure node respectively.

5.4 Contract: the property-enforcing layer

Each contract is a set of essential constraints, extracted from the desired properties of the system model. The constraints are expressed as Boolean equations. The discrete controller is generated to enforce the constraints.

5.4.1 Property 1: no task is active on a failed processor

This property is to ensure that a task τ_i will not be active on a faulty processor ρ_j (i.e., to be in state A^i_j). This is expressed as: $\bigwedge_{\tau_i \in T} \bigwedge_{\rho_j \in P} (a^i_j \wedge (\neg f_j))$. If, in the model system, there exists a transition to a safe state (i.e., one where this property holds), then the synthesis will succeed and the controlled system will always be able to react to a processor failure by moving to a safe state. Otherwise the synthesis will fail, indicating to the designer that her/his system cannot be made fault-tolerant.

5.4.2 Property 2: operate within the bound of $Q1$ of each processor

Property 2 ensures that the cumulated cost of all tasks active on a given processor does not exceed the bound of $Q1$. This property uses the outputs of the criteria calculation node. For active tasks τ_i on all active processors ρ_j, $\sum_{\tau_i \in T} Q1^i_j \leq b_j$.

5.4.3 Property 3: a ready task must transit to the active state

Preventing the tasks from being active would make the sum of Q_1 equal to 0, trivially satisfying Property 2. However it is meaningless to have such systems hence we need to force the ready tasks to be activated by demanding the following expression always to be *true*: $\bigwedge_{\tau_i \in T} (r_i \wedge a^i_j)$.

5.4.4 Property 4: ensure task distribution

This property states that no processor can execute more than one task if there is an active processor executing no task. This is a demonstration of how to achieve simple load balancing. For this, two expressions can not be true at the same time: (1) there is an active processor executing no task; and (2) there is a processor executing more than one task:

For processor ρ_j, the first expression can be written as: $nothingOnProcj \stackrel{def}{=} \neg(\bigvee_{\tau_i \in T} task_i_on_processor_j) \wedge (\neg f_j)$.

For checking if processor ρ_j runs more than one task:

$$\begin{aligned} MoreThan1TaskOnProcj \stackrel{def}{=} & \\ & (task_1_on_proc_j \wedge (task_2_on_proc_j \vee task_3_on_proc_j)) \vee \\ & (task_2_on_proc_j \wedge (task_1_on_proc_j \vee task_3_on_proc_j)) \vee \\ & (task_3_on_proc_j \wedge (task_1_on_proc_j \vee task_2_on_proc_j)). \end{aligned}$$

With these two predicates, property 4 can be expressed as: $\neg(nothingOnProc_j \wedge MoreThan1TasksOnProc_j)$.

5.5 The integration of the system model

Finally, the nodes of the task models, the processor failure model, the criteria calculation node, and the migration indicator node are integrated to the controlled system model along with the contract. The composition of the LTS with the other nodes is shown in Figure 4. Figure 5 illustrates the connection of the nodes in reflect with the Figure 1(b) with the following mappings: $I_u = \{e_j, rc_j, r^i, t^i\}$, $I_c = a^i_j$, and $O = \{a^i_j, s^i\}$.

5.6 Multiple variable optimizations

We can make optimal DCS to minimize or maximize the costs from one state to the next state [12]. We are looking for a controller that can maximize $Q2$ and minimize $Q1$ of

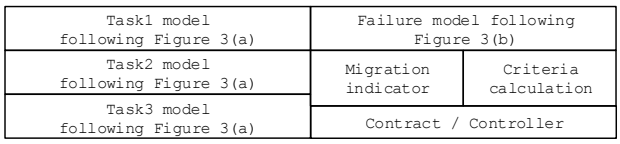


Figure 4: The controlled system model (CSM)

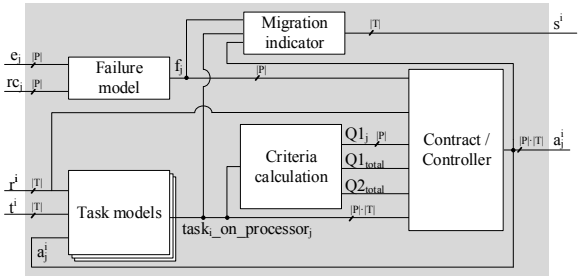


Figure 5: The internal connections of the controlled system model

the target system. There can be several equally weighted solutions, so optimization does not necessarily lead to determinism. SIGALI computes the *maximally permissive solutions* such that the current and future state of the system do not violate the contracts. Figure 6 illustrates the optimization and some possible states for our distributed system. Each state is a configuration of the system, with the convention that the three "blocks" represent respectively processor ρ_1, ρ_2, and ρ_3. For instance, state 8 is the configuration where ρ_1 runs no task, ρ_2 runs τ_3, and ρ_3 runs τ_1 and τ_2.

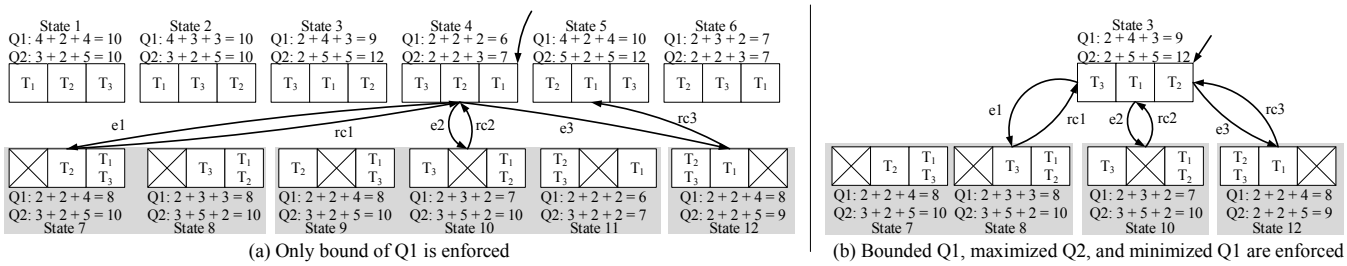
Initially all three tasks are active and none of the processor is faulty. Because of property 4, each processor can only execute one task. In Figure 6(a), where no optimization applies, the controller picks state 4 as the entering state, and chooses states 7, 10, and 12 depending on the failure of ρ_1, ρ_2, and ρ_3 respectively.

The optimizations on multiple variables have to be prioritized. In our example, we first optimize for the maximum of $Q2$, then we look for the minimization of $Q1$. As a result, states 1, 2, 4, and 6 are removed from the entering states. Similarly, state 11 is removed from the potential successors of state 4. After $Q1$ is minimized (Figure 6(b)), only state 3 remains as the entering state. State 7 and state 8 have the same values of $Q1$ and $Q2$, therefore one of them is chosen non-deterministically (because DCS computes the maximally permissive controller). HEPTAGON currently does not provide the constructs for variable optimization performed by SIGALI. To achieve this non-intrusively, i.e., without changing neither HEPTAGON nor SIGALI, we modified the Z3Z file with optimization information as follows.

```
state : [state1, state2]; % the states %
% expression of states, inputs, with constants %
exp1  : state1 and input1;
exp2  : state2 and a_const(1);
target: exp2 or input2; % optimization target %
sys   : ....... % define the system (omitted) %
```

Listing 1: The segment of the original Z3Z file

The generation of the controllers involves using a set of the state variables of the controller. Each time the con-



(a) Only bound of Q1 is enforced

(b) Bounded Q1, maximized Q2, and minimized Q1 are enforced

Figure 6: The behavior of the controlled system

troller takes a step that reacts to the inputs, the state variables evolve (i.e., transit to the next state) depending on the current states, and the inputs. The next states are determined by comparing the values of the optimization targets between the current state and the possible next states to ensure that the contracts are not violated. To achieve this, the states and relevant variables are duplicated for comparisons (`state1`, `state2`, `exp1`, `exp2`, and `target` in Listing 1).

In the Z3Z file, the optimization target is often declared as an expression. Expressions are based on internal variables or other simpler expressions. A tree of expressions can be built, with the root of the tree as the optimization target, and the leaves of the trees are the simplest expressions, e.g., the constants. Duplication of the optimization target involves the duplication of the intermediate expressions.

```
% system with duplicated states as the reference %
sys2     : declare_suff(state_var(sys));
% duplicate the relevant declarations %
exp1__1  : state1__1 and input1;
exp2__1  : state2__1 and a_const(1);
target__1: exp2__1 or input2;
% find the maximum transitions for target %
Strictly_Greater_than(sys,target,target__1,sys2);
```

Listing 2: The Z3Z segment for optimization

Given a system with two states `state1` and `state2` (Listing 1), the optimization goal is named `target`. The results of the duplication process is shown in Listing 2. The state variables are duplicated, and the same applies to the expressions. Duplicated variables are given the suffix `__1`. We then insert the `Strictly_Greater_than` SIGALI function (resp. `Strictly_Lower_than`) to select in the controlled system the transistions that maximize the chosen criterion (resp. minimize). We implemented a tool named CriteriaWrapper to perform the generation of the new Z3Z file automatically.

6. IMPLEMENTATION WITH libDGALS

In the previous section, we have synthesized the controlled system model (CSM), which captures the behaviors of the actual controlled system. To implement the whole controlled system, the model is integrated with the runtime environment which is implemented with libDGALS. The implementation of the system consists of three kinds of libDGALS Clock Domains (CDs): (1) the heartbeat CDs, which interface with the distributed platform by sending the status of its respective host processors; (2) the task CDs, which implement the actual functionalities of the system's tasks; and (3) the controller CD, which wraps the CSM to implement the migration decisions according to the discrete controller. The integration with libDGALS proceeds as follows: the heartbeat and the controller CDs are automatically generated by providing the number of the processors and tasks as the inputs. The programmers only needs to implement the functionality of the tasks' CD. This section details the organization of such integration as well as its implementation details.

6.1 The organization and the operations

Figure 7 illustrates the organization of the system with different numbers of controller CDs. Each processor executes a DGALS program as the run time environment, shown as the rectangles with the grey background, to host DGALS program. To prevent the *single point of failure* that occurs when there is only one controller available in the whole system, each DGALS program is equipped with a controller CD. However for the consistency operations of the controller, only one of the controller CD is acting among them. The election of the acting controller CD happens when the system initializes.

The election process is carried out as follows. Each controller CD starts along with its resident DGALS program and then activates its local heartbeat CD. The heartbeat CDs begin to send heartbeat messages to the remote controller CDs (the ones which are residing on the different DGALS program) to provide them with the necessary information to elect the acting controller CD, shown as Figure 7(b).

The controller CDs are aware of the others and elect the controller CD with the smallest ID to be the new acting controller. Each controller must be assigned a unique ID, chosen arbitrarily by the programmer. The criteria of the election can be changed subject to the characteristics of the system or the decision of the programmer. Only the acting controller CD sends the indication to the local heartbeat CD (Figure 7(a)), and subsequently informs the other controller CDs its existence. When the processor (therefore the DGALS program) of the acting controller CD fails, all the other controller CDs will detect this event due to the misses of the heartbeat messages. Then the election process starts again. Note that there will be no election when there exists an acting controller CD, even if the DGALS program (the processor) with lower ID recovers from its failure state.

The non-acting controller CDs receive and store the state of the acting controller CD, as shown in Figure 7(c), whenever the controller reacts to the inputs. The state of the acting controller CD is used to resume its functionality by the next elected controller when its processor fails. The acting controller CD activates the task CDs on the host DGALS programs according to the decision of its embedded discrete controller. Each task CD sends its context, which consists of the program counter and the working data to all controller CDs, see Figure 7(d). The context of the task CDs are used to perform migration of the task so that the task CD

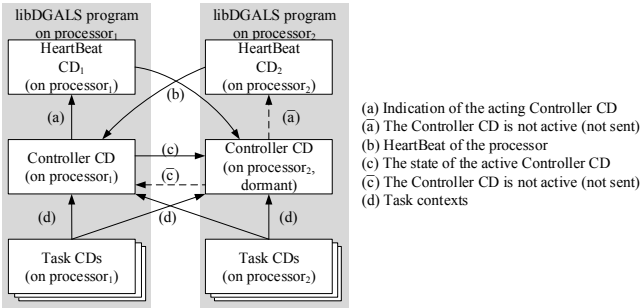


Figure 7: The organization of the system

can resume its execution from the previously stored context (i.e., rollback).

6.2 The internals of the CDs

We refer to Figure 8 in this section, which provides a detailed view of the activities within and between each CD. We define a set of symbols to identify the components of the CDs. In this section we are interested in the local processor ρ_j that executes the controller CD named Controller_j. The remote processors h, where $h = P \setminus \{\rho_j\}$, execute the other controller CDs, Controller_h. Similarly, there is a local heartbeat CD named HeartBeat_j and remote ones named HeartBeat_h to provide processor status to the controller CDs, which in term receive the context from the n tasks (in our case $n = |T| = 3$) available in the system. Because the controller CDs and the heartbeat CDs are parameterized, they are generated *automatically* to ease the programmer's burden.

6.2.1 The heartbeat CDs

The heartbeat CD is activated by the local controller CD (Figure 8(a)). It consists of one ActingControllerID and h HeartBeat reactions. The ActingControllerID receives the ID of the elected controller (Figure 8(b)). If the local controller CD is acting, h heartbeat reactions will notify the other h controller CDs of the existence of the acting controller CD along with the periodic heartbeat messages.

6.2.2 The task CDs

Task CDs are activated on the target DGALS program (i.e., the corresponding processor) by the controller CD (Figure 8(c)). Each task CD can consist of one or more reactions. The reactions are mapped to threads, each reaction has its own context. The context of a task CD consists of the program counter of the corresponding thread, and a set of working data (the structure on which the reaction operates). The *TaskContext* reaction collects the context of each sub-behavior reaction and sends the collection to the each controller CD (Figure 8(d)). The context of a task CD is used for the task migrations. When a task CD migrates, i.e., when the controller issues the a^i_j signal in the controlled system model, the task CD will be activated on the target DGALS program with the latest context that was collected previously. The TaskContext reaction then dispatches the context of individual sub-behaviors to the corresponding reactions to resume their executions (Figure 8(e)). When a task CD migrates, the task CD will be terminated first and will resume its execution on the target processor.

6.2.3 The controller CDs

As the result of DCS, C code files are generated, representing the functionality of the controlled system model. The system can be invoked with a *step function* to perform the behavior of the current logical time as in the synchronous MoC, as a subset of the libDGALS. It is straightforward to wrap the controlled system model in the reaction Controller, as a part of the controller CD (Figure 8(f)). In the Controller CD, the Initialize reaction first elects the acting controller CD according to the status of the other h processors from the heartbeat messages. Such messages are received by HBDetector_h reactions through the channels $\text{cHB}_h\text{ToCtrl}_j$ (Figure 8(g)). The received messages is interpreted and used to signal the Initialize reaction through $\text{sCtrl}_h\text{Alive}$ (Figure 8(h)). Once the initialization completes, the reaction Controller starts, together with the *ActiveTask* reaction (Figure 8(i)). The Controller reaction forwards e_h, rc_h, r^i, and t^i to the discrete controller (Figure 8(j)). For example, reaction ActivateTask checks if the binary of the task τ_i is available and issues the r^i to the controller reaction to make the task τ_i (in the system model) transit to the ready state. Similarly, the reactions *TaskTermination* (one for each task) receive the notification of the termination from the Task CD (Figure 8(k)) and dispatch the t^i signal to the controller. If the heartbeat of a processor is missing, the reactions FaultReport will send e_h to the Controller reaction, likewise rc_h will be sent when the processor recovers. Because the discrete controller operates on a set of the state variable which can be considered as the context of the controller, whenever the context of the controller changes, i.e., the controller advances its steps, the latest context is sent to the non-acting controller CDs through the h SendCtrlInfo_h reactions (Figure 8(l)). As a counterpart, the non-acting controller CDs receive the context of the acting controller via one (i.e., the acting controller) of the h ReceiveCtrlInfo_h reactions (Figure 8(m)). The context of the controller is used to resume the operation of the newly elected controller when the previous acting controller fails.

7. RELATED WORKS

Formal approaches to the design for fault-tolerant systems have mostly considered the problem of verification, for instance in the context of process algebra [3]. They verify that an existing, hand-made design satisfies a certain equivalence with the nominal functionality specification in case of faults. In contrast, DCS approaches [10] synthesize automatically a controller that will insure this by construction. Planning under uncertainty is another existing approach [10], so far only demonstrated with 1-fault tolerant paths. We place ourselves in the framework of reactive systems and LTSs. Moreover, we tolerate several failures, not only one. In contrast to a relevant work that uses DCS for distributed controller[6], here we synthesize a centralized controller but replicate it on each processor therefore making the controller itself fault-tolerant and preventing the existance of a single point of failure in the deployed system. DCS have been recently used for the control of computing tasks, on dynamically reconfigurable FPGA [2], or for the coordination of managers on autonomic systems [8]. However these works do not consider the distribution of the controller itself, which is necessary when considering fault-tolerance.

8. CONCLUSIONS AND FUTURE WORKS

We have shown the flow of modeling, synthesizing, and im-

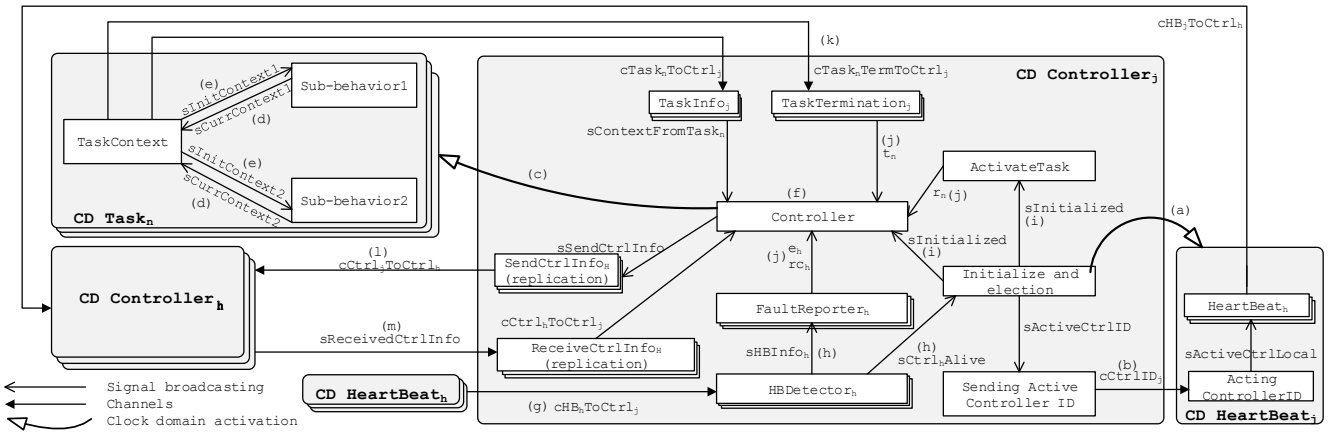


Figure 8: The details of the CDs, their internals, and their relationships.

plementing a fault-tolerant system with a formal approach. The system is modeled with the HEPTAGON synchronous language, and the synthesizing of the discrete controller is achieved through the use of the SIGALI DCS tool. From the point of view of implementing a fault-tolerance system, our approach is interesting in the sense that the system is integrated with a runtime support based on libDGALS, which provides the essential features to implement the task migrations and other dynamic fault-tolerant features, such as the failure of the processor where the acting controller resides. To the best of our knowledge, this is the first framework able to provide a complete implementation of a fault-tolerant distributed system, where the correctness of the fault-tolerance is guarenteed by a formal method (DCS), the controller itself is protected from processor failures, and several criteria can be taken into account in the DCS procedure to optimize aspects such as processor load or quality of service.

Interesting perspectives include the following aspects: (1) even though we use a generic criteria ($Q1$ and $Q2$) in this paper, they can be substituted with the WCET of the tasks, or the power consumption of the processors; (2) weights could also be associated to transitions in the overall system model, therefore allowing us to take into account the migration cost in our optimal DCS procedure.

9. REFERENCES

[1] K. Altisen, A. Clodic, F. Maraninchi, and E. Rutten. Using controller-synthesis techniques to build property-enforcing layers. In *ESOP*, pages 174–188. Springer, 2003.

[2] X. An, E. Rutten, J.-P. Diguet, N. L. Griguer, and A. Gamatié. Autonomic management of dynamically partially reconfigurable FPGA architectures using discrete control. In *10th International Conference on Autonomic Computing (ICAC'2013), San José, CA, USA*, pages 59–63, 2013.

[3] C. Bernardeschi, A. Fantechi, and L. Simoncini. Formally verifying fault tolerant system designs. *The Computer Journal*, 43(3):191–205, 2000.

[4] G. Berry and L. Cosserat. The synchronous programming language esterel and its mathematical semantics. In *Seminar on Concurrency*, volume 197, pages 389–448, 1984.

[5] G. Delaval, E. Rutten, and H. Marchand. Integrating discrete controller synthesis into a reactive programming language compiler. *Discrete Event Dynamic Systems*, pages 1–34, 2013.

[6] E. Dumitrescu, A. Girault, and E. Rutten. Validating fault-tolerant behaviors of synchronous system specifications by discrete controller synthesis. In *WODES*, 2004.

[7] A. Girault and E. Rutten. Discrete controller synthesis for fault-tolerant distributed systems. In *Proc. Ninth Int. Workshop on Formal Methods for Industrial Critical Systems, FMICS*, 2004.

[8] S. M.-k. Gueye, N. De Palma, E. Rutten, A. Tchana, and D. Hagimont. Discrete control for ensuring consistency between multiple autonomic managers. *Journal of Cloud Computing: Advances, Systems and Applications*, 2(1):16, 2013.

[9] C. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, 1978.

[10] R. Jensen. DES controller synthesis and fault tolerant control-a survey of recent advances. *The IT University of Copenhagen*, 2003.

[11] M. Le Borgne, H. Marchand, E. Rutten, and M. Samaan. Formal verification of signal programs: Application to a power transformer station controller. In *Algebraic Methodology and Software Technology*, pages 271–285. Springer, 1996.

[12] H. Marchand, O. Boivineau, and S. Lafortune. Optimal control of discrete event systems under partial observation. In *Decision and Control*, volume 3, pages 2335–2340. IEEE, 2001.

[13] H. Marchand and M. Samaan. Incremental design of a power transformer station controller using a controller synthesis methodology. *Software Engineering, IEEE Transactions on*, 26(8):729–741, 2000.

[14] P. J. Ramadge and W. M. Wonham. Supervisory control of a class of discrete event processes. *SIAM journal on control and optimization*, 25(1):206–230, 1987.

[15] W.-T. Sun, A. Girault, Z. Salcic, and A. Malik. libDGALS: A Library-based Approach to Design Dynamic GALS Systems. In *SIES 2014*, 2014.