Modeling and Verification of Redundancy Policies*

Hamza Chouh, Charlotte Callon, Ghita Jalal
Frédéric Boulanger, and Safouan Taha

Supélec E3S – Computer Science Department
firstname.lastname@supelec.fr

Abstract. In this paper, we present a metamodel for specifying redundable software and hardware architectures. This metamodel takes into account the constraints on the number of redundant elements, the number of allowed failures, the execution times and allocation constraints. From such a specification, we generate all possible structural configurations. Then, we check that each of these configurations can be scheduled. This has been implemented as a tool chain relying on Alloy, SynDEx, and model transformations in Eclipse/EMF. This work allows system architects to explore different hardware and software architectures to implement different redundancy policies. It has been applied on a simple case study from the Ariane V launcher.

1 Introduction

A large number of fields such as industry, medicine and transport have a need for highly complex systems that are required to provide their functionalities under constraints of safety and reliability. Since computer systems and software are increasingly used in such critical systems, it is well known that both software and hardware are likely to present some defects that can lead to failures. Handling these defects without any catastrophic consequences is required when there is no guarantee of removing all the defects from the system design. A way to design such systems is to provide redundancy, that is to duplicate some critical parts of hardware and/or software. Hardware redundancy is a common practice, but software redundancy is also essential since a software execution can be corrupted by, for example, memory faults.
In this paper, we first define a meta-model that helps designing redundant systems. Our meta-model provides the necessary constructs to specify both hardware and software architectures together with allocation and timing constraints. Then, we give a tool chain to exploit such abstract models to generate concrete redundant configurations.
The first step to model the redundant system is to give a high-level description of its hardware and software architectures in terms of components, without taking redundancy into consideration. This description can be thought of as a functional decomposition of the system. Then, to

* This work has been supported by CNES (http://www.cnes.fr).

provide information about redundancy, the user will be able to specify the following constraints:

Policy constraints: choice of cold, warm or hot redundancy, number of copies of a hardware or a software component, number of failures the system must support;

Allocation constraints, or how software components can be mapped onto hardware components;

Communication constraints between software components that are related to control or data exchanges;

Timing constraints about both the communications over the buses and the execution of the software components on the hardware components to which they may be allocated.

From this user model, our tool chain will generate (if any) hardware and software configurations that satisfy all the constraints on policy, allocation and communication. These concrete configurations instantiate the abstract model with precise information about replication of hardware and software components, allocations, connections to buses etc. Then, the tool chain will take the timing constraints into account to check which configurations can be scheduled and therefore executed. It will also provide a valid schedule of software executions on the available hardware.

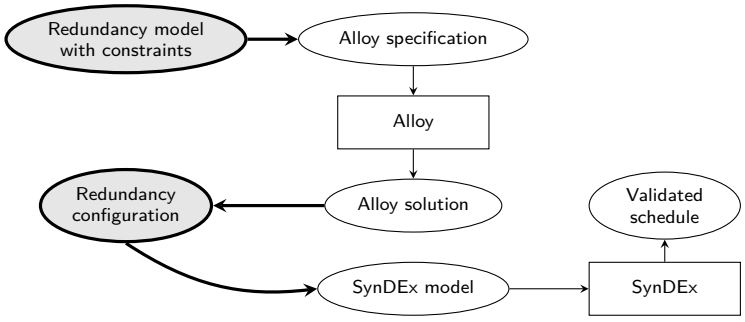

Fig. 1. Tool chain and work flow

Our tool chain, shown on Fig. 1, relies on the following parts:

1. The `Redundancy Metamodel` for describing the HW/SW architecture, the redundancy policy as well as allocation, communication and timing constraints,
2. A model transformation from the `Redundancy Metamodel` toward the *Alloy*[1] language to obtain a set of *Alloy* predicates that are solved to generate concrete configurations,
3. A model transformation from the *Alloy* output solutions back to the `Redundancy Metamodel` to analyze the generated configurations,
4. A model transformation from the `Redundancy Metamodel` that produces *SynDEx*[2] code for checking if a concrete configuration can be scheduled.

2 Simple Case Study

To show how redundant systems can be modeled using our meta-model, we will go through an example which is a very simple model of the Ariane V flight control-command system.

2.1 System architecture

For the purpose of this case study, we decomposed the Ariane V hardware system as follows:
- **SRI** (Inertial Reference System): the sensors,
- **OBC** (On-Board Computer): the processors,
- **EPH** (Hydraulic flight control Electronics): the actuators.

Similarly, we decomposed the software system as follows:
- **Measures**, getting the position and attitude of the launcher.
- **Computations** of the necessary actions for following the trajectory.
- **Commands**, such as increasing thrust or steering.

2.2 Policy

Now that we have a functional decomposition of our system, we need to choose the type of redundancy - cold, warm or hot - and the number of copies for each component. For our example, the following redundancy policy can be chosen for the hardware:
- **Hot** redundancy for sensors: all sensors are measuring all the time,
- **Warm** redundancy for processors: all processors are initialized but only one of them has the control over the system,
- **Cold** redundancy for actuators: only one actuator is running, the others are disconnected.

The redundancy policy for software:
- **Hot** redundancy for measures,
- **Warm** redundancy for computations: several types of algorithms will be running but only one will provide results,
- **Cold** redundancy for commands.

For this example, we choose to have two copies of each component of the system and we want to handle one failure for each duplicated component.

2.3 Allocation

Allocation consists in mapping each software component onto a hardware component. The meta-model we designed allows the allocation of several software components on several hardware components. For instance, we could choose to allocate measures **and** computations on **processors** or **sensors**, so that a measure could be done indifferently on a processor or a sensor. The same would go for computations. This can help covering more cases.
For our example of the Ariane space launcher, we keep things simple and allocate:
- Measures on sensors,
- Computations on processors,
- Commands on actuators.

2.4 Communication and Timing

We know that the **computations** depend on **measures** and produce **commands**. We thus add data dependencies and the size of the corresponding data (to compute communication time). We also add information about the period of software tasks and their execution time on the given hardware, to obtain the final model shown on Fig. 2.

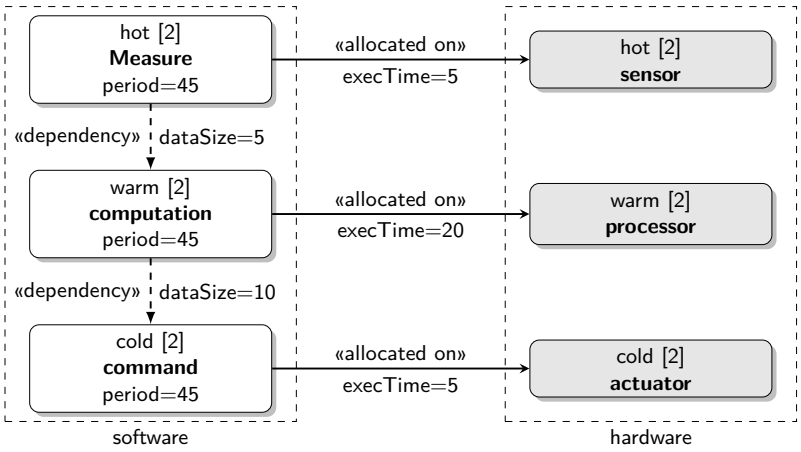

Fig. 2. Ariane V redundancy model

3 Related Work

Several approaches of redundancy can be found in the literature. Many focus on the automatic generation of fault-tolerant schedules, using assumptions on the failure model (fail-silent or fail-safe), on the homogeneity of the hardware (same execution time for a task on all processors, same communication time for a piece of data on all buses) or on the number of supported faults [3–5]. Other approaches dynamically schedule backup tasks when a primary tasks fails [6]. Our approach aims at offering analysis tools to the designer of a system and of its redundancy policy. We therefore rely on previous work about fault tolerance and redundancy and try to harness existing tools in a tool chain. The metamodel we created for modeling the hardware and software components of a system and the redundancy constraints that they should satisfy, as well as the model transformations between this metamodel and the tool specific formats, allows us to share information among these tools and to exploit their results. We support different execution times on different processors and different communication times on different buses, as in [7]. The meta-model we propose here is inspired from the MARTE profile [8] (GRM and HLAM packages) and we added an extension for

supporting abstract allocation constraints since MARTE's Alloc package only models static allocation.

4 Redundancy Metamodel

As shown in the left part of Fig. 3, our *Ecore* metamodel provides the user with a concise specification language to model the hardware/software architectures together with policy, allocation, communication and timing constraints. For this purpose, the Redundancy Metamodel provides the following concepts :

Redundable a type of software or hardware elements that will be redundant. It has a maximum number of instances, a maximum number of failures and a policy type.

Allocation a mapping constraint between a software cluster and a hardware cluster (e.g. a measure must be executed on a sensor). It has an `executionTime` parameter to set the specific execution time of an instance of `sw` when allocated on an instance of `hw`.

Dependency a need for communication between two software clusters (e.g. the need for an algorithm to get data from sensors). It has a `dataSize` parameter that models the size of the data that is sent from the `src` cluster to `dst`.

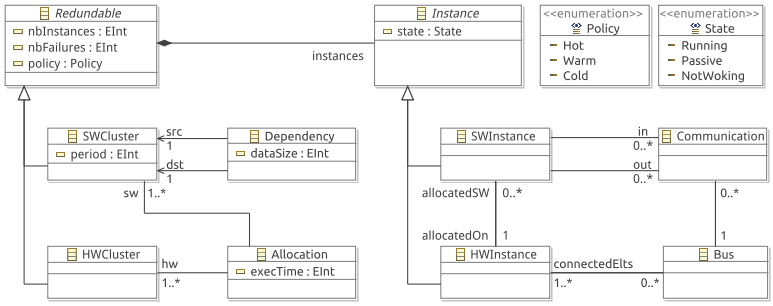

Fig. 3. Redundancy Metamodel

A typical example of a user model for a redundant system was given in Fig. 2.

Our tool chain generates, from such a high-level redundant system model, many concrete configurations with precise connections to buses and precise allocations between software and hardware instances. As shown in the right part of Fig. 3, our Redundancy Metamodel provides the following low level concepts to model these concrete configurations :

Instance an element that will be part of the system and directly derive from a cluster. Each hardware or software instance has a state to describe whether it is running, passive or failed.

Bus a communication medium between hardware instances.

Communication a communication between two software instances. It instantiates a dependency between software clusters.

An example of concrete configuration written in our meta-model is shown on Fig. 5 in section 5.

The first usage of the redundancy metamodel is the specification of redundant systems at a high level of abstraction. The second usage of the redundancy metamodel is to provide a way to generate, view and modify concrete configurations. From a high level description, we generate concrete configurations automatically using *Alloy*.

5 Solving Structural Constraints using *Alloy*

To generate sets of hardware and software configurations that fulfill all the requirements induced by the constraints from the high level redundancy model, a constraint solver is needed. We chose to use *Alloy* [1] for this purpose since it is very well adapted to constraint solving over classes and object-oriented models. *Alloy* is the name of a constraint modeling language and of the associated constraint solver, both developed by the M.I.T.'s Software Design Group.

We wrote a set of *Alloy* classes and predicates that match the elements of our redundancy metamodel (e.g. SWCluster, SWInstance...). Then, we developed a model to text (M2T) transformation based on Java to translate a redundant system model to the equivalent *Alloy* declarations. We show in the left of Fig. 4 some of the classes and predicates we wrote, and in the right of the figure some of the generated *Alloy* code for the launcher case study. Finally, the whole code is given to the *Alloy* solver.

```
abstract sig Redundable {
  instances: some Instance,
  nbInstances : one Int
  ...
}
...
sig AllocationConstraint {
  sw: one SWCluster,
  hw: one HWCluster,
  execTime: one Int
} {
  all i: sw.instances |
      i.allocatedOn.class in hw
  ...
  execTime >= 0
}
...
```

```
sig SRI  extends HWCluster  {}
sig OBC  extends HWCluster  {}
sig EPH  extends HWCluster  {}
sig Measure  extends SWCluster  {}
sig Computation  extends SWCluster  {}
sig Command  extends SWCluster  {}
...
fact allocationConstraints {
  allocationConstraint[SRI, Measure]
  allocationConstraint[OBC, Computation]
  allocationConstraint[EPH, Command]
}

run compute for
    exactly 2 SRI_,
    exactly 2 Measure_
    ...
```

Fig. 4. Example: Alloy specification

To illustrate the use of *Alloy* for solving the constraints of a high level redundancy model, we over-constrained our case study:

- Two different computations must be running (hot redundancy),

– These two computations must be allocated on *different* processors. We recall that there are two processors and that one may fail.

Running our M2T generator on the corresponding model and then running *Alloy* on the generated code leads to an error message from *Alloy* that indicates that no configuration could be found.
From this error, we can conclude that our variant of the launcher system is inconsistent. We therefore modify it by allowing the two computation programs to be allocated on the same processor and keeping the other constraints unchanged. Running *Alloy* on the model produced by our tool chain gives us the configuration shown in Fig. 5 (in which only instances and allocations are shown).

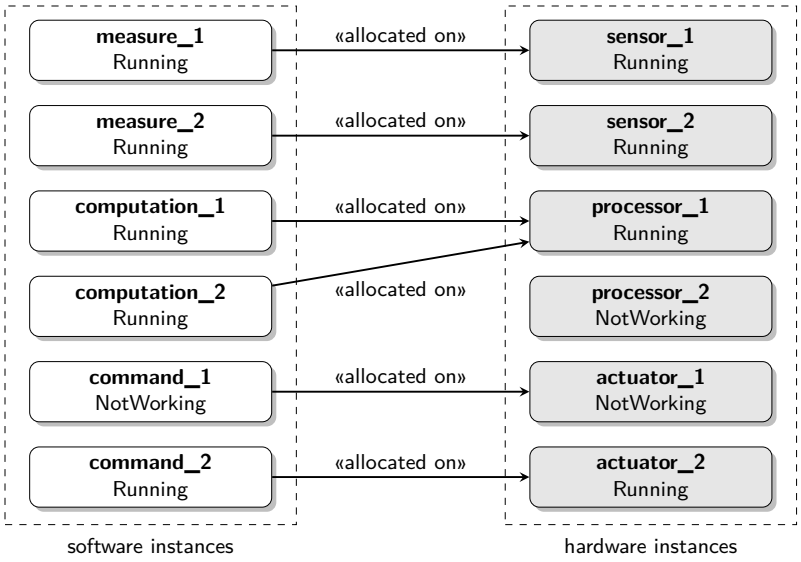

Fig. 5. Example: a generated configuration

Now that we have consistent configurations fitting all the constraints that were specified using the Redundancy Metamodel, we need to check their schedulability without having to manually write code for a schedulability checker.

6 Checking Timing Constraints using *SynDEx*

Running *Alloy* gives an allocation configuration that complies with the system architecture and its structural constraints (policy, allocation and communication constaints). It remains that timing constraints should be taken into account to ensure the schedulability. We use *SynDEx* [2, 9], a software developed by the INRIA Paris-Rocquencourt Research Center

in France. *SynDEx* allows us to model the timing constraints on interconnected systems and to check their schedulability. The *SynDEx* model is composed of three descriptions:

- The description of the software **application** as a set of interconnected nodes. Each node represents a computation which can be run only when its input data is present at the entry ports. The connections between nodes represent data dependencies.
- The description of the hardware **architecture**, which contains two types of elements: media and operators. Operators have communication ports and processing time constraints, and media have data transportation time constraints.
- The description of the **allocation** of the application onto the architecture with timing constraints.

From the configuration generated by *Alloy*, we get back to a more detailed model based on our meta-model using an ANTLR-based compiler we wrote. Using this model and the timing constraints initially entered using the redundancy metamodel as inputs, another Java-based M2T transformation in our tool chain creates a *SynDEx* representation of the model. Then we ask *SynDEx* to generate a schedule. If *SynDEx* returns a schedule, we know that the configuration is schedulable and correct. Else, we can analyze the result of *SynDEx* to understand why the scheduling fails.

If we consider again the model of the launcher depicted in Fig. 2, and if we add hot redundancy for computations, *Alloy* finds an allocation configuration such as the one of Fig. 5. However, when run through *SynDEx*, it appears that this configuration is not schedulable.
We may try to run *SynDEx* on some other redundancy configurations generated by *Alloy*. However, in this case, the inconsistency is actually located in the constraints we have chosen. Indeed, two computations must be able to run on the same remaining processor (when one processor fails) and to send data to actuators within a period of 45 cycles. Sending data requires 10 cycles and computing requires 20. This is why *SynDEx* is unable to schedule such a cycle: $(20 + 10) * 2 > 45$.
By allowing one of the two computations not to run (warm redundancy), *Alloy* generates a configuration that *SynDEx* is able to schedule, and we obtain the final scheduling shown on Fig. 6. The concrete configuration generated by *Alloy* in this case runs one instance of the computation on the remaining processor after the failure of the first processor.

7 Discussion

The tool chain we developed provides a mean to model a system and redundancy constraints over it. Then, it generates allocation configurations that respect these constraints. Finally, the schedulability of these configurations can be checked. However, this work is a first approach to the modeling and verification of redundancy policies and has some limitations we are aware of.
First, our meta-model does not consider the cost of reconfigurations between different allocation configurations. We only take into account the

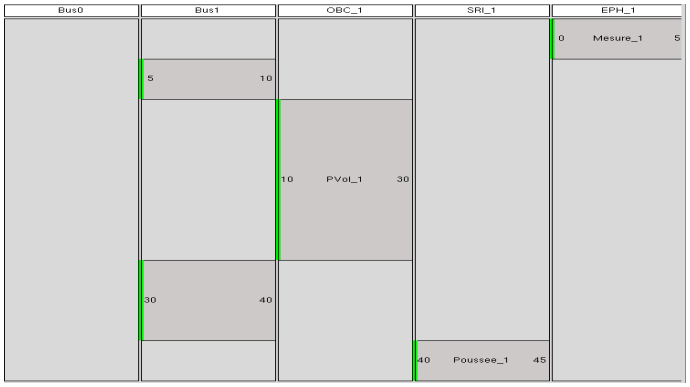

Fig. 6. Scheduling example

execution time of each software task when checking schedulability, and in some cases, this is not enough. Indeed, considering the example of Ariane V where OBCs (On-Board Computer) have warm redundancy policy, if one of the OBCs fails, some time is needed for the redundant one to be ready to take over, during which the system is unable to perform computations. This is not necessarily fatal but it needs to be modeled. More generally, going from an allocation configuration to another takes some time that may have consequences on the behavior of the system and is important to model.
In addition, we started from the hypothesis of a fail-stop system, which means that we only consider failures that are detectable. Tolerance to silent failures is a complex problem that we did not address in this work. Moreover, the generated redundancy configurations are not sorted, which means that some configurations could be better optimized from a schedulability point of view when others consume less resources. This is partially linked to the fact that no qualitative constraints are given at the generation step.

8 Conclusion

We have presented a metamodel for redundancy modeling and a tool chain for generating and verifying configurations. The main contributions of this paper are:

- The metamodel to express redudancy both at a high level using constraints, and at a lower level using detailed allocation.
- The model transformations for building an *Alloy* model from the redundancy model, and for building an allocated redundancy model from the configurations generated by *Alloy*.
- The model transformation from an allocated redundancy model to a *SynDEx* model in order to check its schedulability.

The main motivation of this work was to ease the exploration of different redundancy policies for a system by verifying automatically if a redundancy model is sound (are there configurations that satisfy the redundancy constraints), and if the configurations that satisfy the constraints can be scheduled.

Future Works

The next steps of this work will improve our redundancy tool chain by adding the previously discussed features. We will add information about reconfiguration times to take into account the dynamic aspects of redundancy, and we will provide a quality function to evaluate the relevance of a generated configuration, which could, for example, help minimizing the costs or improving the safety of a system.

References

1. Daniel Jackson. *Software Abstractions.* The MIT Press, September 2011.
2. T. Grandpierre, C. Lavarenne, and Y. Sorel. Optimized rapid prototyping for real-time embedded heterogeneous multiprocessors. In *Proceedings of 7th International Workshop on Hardware/Software Co-Design, CODES'99*, Rome, Italy, May 1999.
3. Alan A. Bertossi, Luigi V. Mancini, and Federico Rossini. Fault-tolerant rate-monotonic first-fit scheduling in hard-real-time systems. *IEEE Trans. Parallel and Distributed Systems*, 10:934–945, 1999.
4. Xiao Qin, Zongfen Han, Hai Jin, Liping Pang, and Shengli Li. Real-time fault-tolerant scheduling in heterogeneous distributed systems. In *in Proceedings of the 2000 International Conference on Parallel and Distributed Processing Techniques and Applications, Las Vegas*, pages 26–29, 2000.
5. Krithi Ramamritham. Allocation and scheduling of precedence-related periodic tasks. *IEEE Trans. Parallel Distrib. Syst.*, 6(4):412–420, April 1995.
6. Marco Caccamo, Giorgio Buttazzo, and Scuola Superiore S. Anna. Optimal scheduling for fault-tolerant and firm real-time systems. In *In 5th International Conference on Real-Time Computing Systems and Applications. IEEE*, pages 223–231, 1998.
7. A. Girault, H. Kalla, M. Sighireanu, and Y. Sorel. An algorithm for automatically obtaining distributed and fault-tolerant static schedules. In *Proceedings of International Conference on Dependable Systems and Networks, DSN'03*, San Francisco, California, USA, June 2003.
8. Safouan Taha, Ansgar Radermacher, Sebastien Gerard, and Jean-Luc Dekeyser. An open framework for detailed hardware modeling. In *SIES*, pages 118–125. IEEE, 2007.
9. A. Vicard and Y. Sorel. Formalization and static optimization for parallel implementations. In *Proceedings of Workshop on Distributed and Parallel Systems, DAPSYS'98*, Budapest, Hungary, September 1998.