Towards Seamless Integration of N-Version Programming in Model-Based Design

Tingting Hu*, Ivan Cibrario Bertolotti†, Nicolas Navet*
* University of Luxembourg, 6 Avenue de la Fonte, L-4364 Esch-sur-Alzette, Luxembourg
Email: {tingting.hu, nicolas.navet}@uni.lu
†National Research Council of Italy – IEIIT, c.so Duca degli Abruzzi 24, I-10129 Torino, Italy
Email: ivan.cibrario@ieiit.cnr.it

***Abstract*—The ever-growing complexity of present-day software systems raises new and more stringent requirements on their availability, pushing designers to make use of sophisticated fault tolerance techniques far beyond the areas they were traditionally conceived for, and bringing new challenges to both the modelling and implementation phases. In this paper, we propose a design pattern to model in a domain-specific language one of the prominent fault-tolerant techniques, namely the N-version programming. It can be integrated seamlessly into existing applications to enhance their functional correctness, while still preserving the timing characteristics, in particular the sampling times. Besides, it is also designed in a way to ease the automatic code generation. A counterpart of the same framework is also implemented in a lower-level programming language, for use when direct model execution is impractical, like in severely resource-limited embedded targets.**

***Index Terms*—Model-based design, Fault-tolerance, Industrial cyber physical systems**

I. Introduction and Related Work

As industrial Cyber Physical Systems become more and more software intensive, software defects tend to become the major source of faults, which impairs control quality and may even lead to severe consequence. This foresees a strong need of the adoption of suitable software fault-tolerant techniques to improve system availability, reliability and dependability. Introducing software diversity and redundancy proves to be an effective mechanism towards this goal. Although most of the seminal work on software fault-tolerance methods has been performed in the late 70s, with the introduction of *recovery blocks* (RB) [1] and *N-version programming* (NVP) [2], research continues today, with further theoretical advances and their application to specific domains [3].

Although NVP is one mainstream technique that has been deployed in many mission-critical applications, it has been criticized for not being cost effective from both the time and effort point of view for what concerns deployment. This also introduces difficulties to investigate its effectiveness and its impact on the system during the *design* phase, further aggravating modeling challenges that are already complex by themselves. What is still lacking, to the extent of the authors' knowledge, is a way of relieving as much as possible designers and programmers from the effort of introducing fault-tolerance elements in their work, as they go from design to implementation. This is especially true since nowadays there is a stronger need to incorporate fault-tolerance even in resource-constrained embedded systems. Moreover, Model-Based Design is steadily growing in popularity and now demands a better integration of non-functional concerns, like dependability, in the design flow. This work, which lays the foundations for automating the addition of safety mechanisms into an application, is a contribution in that direction.

In this paper, we propose a universal approach to model NVP with the Cyber Physical Action Language (CPAL) [4], which can be integrated seamlessly with existing system models to enhance them with fault-tolerance features. Most importantly, it is a general solution that can be borrowed and re-used. This releases system designers from inner details and burden of fault-tolerance theory and practice, so that they can just focus on the application logic. *Modularity* is another essential principle we followed during framework design. In particular, the modelling of functional behavior (e.g. member version logic) and non-functional characteristic (e.g. fault-tolerance) are kept apart and isolated as much as possible.

Unlike its higher-level counterparts, namely modelling fault-tolerance by means of constructing mathematical models [5], our approach sheds more lights on details about implementation, up to the point of driving a fast implementation of NVP with a low-level programming language. At the same time, it still keeps a consistent link between model and implementation, unlike lower-level approaches do. For instance *Hystrix* [6] provides a comprehensive fault tolerance framework. However, besides having no direct links to any modeling language, it is Java-centered and its main focus is on large, distributed consumer systems. Other execution frameworks [7] achieve high efficiency by means of binary code rewriting techniques, which makes them not readily adaptable to the diverse processors adopted in embedded systems, and may complicate interaction with modeling tools and languages.

The paper is organized as follows: Section II recalls the key concepts of NVP while Section III presents the main features of the CPAL language, which is adopted in Section IV to illustrate the full-fledged NVP modeling framework. Even though models written in CPAL can run directly on certain type of embedded platforms [4], we present in Section V a NVP implementation based on a widely used low-level

programming language (namely C) derived from the high-level model. This implementation can be used when there is no sufficient resources to accommodate direct model interpretation. Finally, in Section VI we draw some conclusions.

II. N-Version Programming

The original concept of NVP implements the N-fold replication of the same computation, carried out by means of N software modules, called *member versions* (the same terminology and abbreviations as in [2] are used here for consistency). Member versions are executed concurrently and operate on the same inputs. At predefined *cross-check points* (cc-points) they generate *comparison vectors* (c-vectors), that is, a representation of their internal state. A *decision algorithm* compares the c-vectors corresponding to each cc-point and checks whether or not they are in agreement, while also determining the overall NVP output. Other outcomes of the decision are a feedback towards member versions (for instance, to terminate a faulty version) and recovery actions.

Even restricting the scope to decision algorithms based on software voting, many approaches have been proposed in the literature [8]. Among them *majority voting*, on which we focus in the following, is in widespread use. Regardless of how it is realized, the decision algorithm may form a single point of failure, and hence its availability and safety are of great importance. However, this issue can be tackled by means of appropriate software design and implementation techniques discussed, for instance, in [9]. Even more importantly, this aspect does not affect the way NVP is modeled, at the level of abstraction typical of the most common modeling tools.

To enhance diversity and reduce the probability of common-mode failures, member versions may be independently designed and implemented from the same initial set of requirements, which must also define cc-points location, c-vectors content and format, the decision algorithm to be used, and responses to decision outcomes. As a further protection against other kinds of software and hardware faults, they may also be developed using dissimilar programming languages and deployed on distinct nodes of a distributed system.

Within these general principles, several NVP variants have been introduced and can be adopted to protect against both software and hardware faults. For instance, if the same software is replicated for all member versions, but member versions are deployed on distinct nodes, through spatial diversification it will protect also from hardware faults affecting those nodes. Accordingly, the modeling strategy must be flexible enough to support and accommodate these variants with minimal user involvement.

III. The CPAL Modelling Language

CPAL [4] is a versatile language that can be used to model, simulate, and program embedded systems, encompassing both their functional and non-functional behavior, especially timing and predictability concerns. At the same time, the syntax of CPAL is close enough to the C language, which helps smoothen the learning curve sometimes associated with other approaches, for instance, synchronous languages [10]–[12]. It supports all basic data types found in typical programming languages within a strong type system. For further expressiveness, it also offers more sophisticated native types like *unsized arrays* (variable-length counted arrays used as function and process parameters) and *communication channels*. Channels have both FIFO *queues* and LIFO *stacks* as sub-types, thus providing flexible but well-defined access policies.

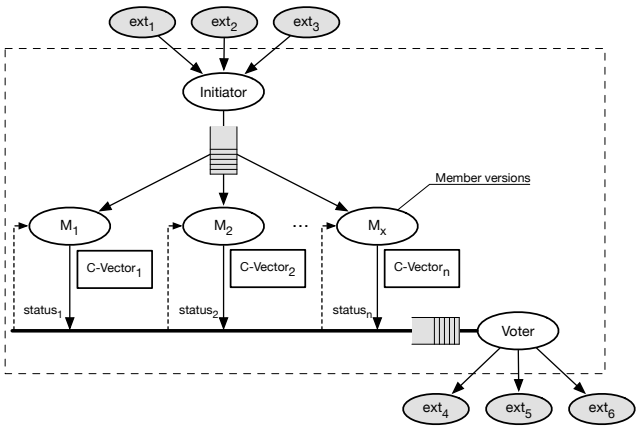

Fig. 1. The NVP modeling framework.

A central element of CPAL is the *process*, which encapsulates a *Mealy* Finite State Machine (FSM)—that is, a FSM in which transition conditions are evaluated and possibly taken before executing the code of the target state. By making use of another native CPAL concept, *time*, processes can be instantiated and scheduled for periodic execution. Event-triggered systems can be described, too, by specifying process instance *activation conditions*, which may work alone or be considered together with the *period* of an instance. A process in CPAL can be instantiated in the following way.

```
process Proc_Type: p[period,offset][conditions](parameters);
```

if the first release of a process should not take place at time zero, but at a specific time after time 0, an *offset* can be indicated next to the period.

Another notable language feature is that, unlike other modeling languages, it can be executed not only within a simulation environment, but also on a real target platform by means of an interpreter. This has the twofold goal of maintaining a strong link between the modeling and the implementation/execution phases, while decoupling the execution platform from the application to verify the correctness of both independently. Besides natively providing several, non-preemptive scheduling policies on a single computing resource, CPAL also supports true concurrency in simulation mode, by using a dedicated interpreter per computing resource and properly synchronizing them.

IV. The NVP Modeling Framework

Fig. 1 depicts the overall NVP modeling framework. It consists of three kinds of component, that is the *initiator*,

member versions and the *voter*, which are modeled as independent concurrent processes in CPAL. Center to the figure is the member versions, which encapsulate alternative algorithms that realize the same goal defined in the initial specification. In other words, member versions are supposed to operate on the same inputs and generate outputs of the same kind. Correspondingly, each of them exports a comparison vector and an execution status indicator (conclude its computation or not) to the voter in suitable form.

When the comparison vectors from different member versions are available, the voter starts processing them according to the decision algorithm embedded in it. More specifically, the execution of the voter will be triggered depending on the member version execution status, whereas the comparison vectors, which includes the outputs calculated by a member version and other state information, will be communicated to the voter by means of *queues*. The adoption of queues permits to model NVP in a uniform way, regardless of how its components will be deployed in practice, on a single node or in a distributed manner.

Depending on the outcome of the decision algorithm (which is generally implemented as certain kind of voting algorithm), a member version may be considered as faulty and terminated by the voter. This corresponds to the dashed arrows going from the voter to the member versions in the figure.

The initiator is a new component introduced for the convenience of assisting modeling. Since one main goal of the proposed framework is to allow system designers to explore fault-tolerant technique(s) at early design phase, for instance by patching it to different modules (represented as processes) of the modeled system, it is important that the way the patched module(s) interacts with other parts (e.g., ext_1, $\ldots ext_6$ in Fig. 1) of the system do not change. The initiator serves exactly this purpose by grouping together the inputs the module originally works on and dispatching them to the member versions. Instead, the outputs will be the ones produced by the voter and be provided to the remaining parts of the modeled system. Similarly, queues are used for the communication between the initiator and member versions because they not only relax constraints on the type and number of data being provided to member versions but also provide a convenient way for synchronization. In the following, the module of a modeled system that is to be enhanced with NVP will also be referred to as the *original process*.

The number of member versions is *configurable* at will, even though two or three member versions are most commonly seen. In particular, three member versions guarantee the correctness of results in presence of a single faulty member version. This case will be considered in the example demonstrated below. System designers are just required to implement the application-specific algorithms to be used within each member version, and fill them into the skeleton of the full-fledged NVP model defined in this framework, a process that can easily be fully *automated* by means of code transformation techniques. The following sections will zoom in into individual NVP components, while the code of the NVP is freely available in the CPAL examples library at https://www.designcps.com/cpal-code-examples-index/.

```
1  processdef Initiator(in uint32: a, in uint32: b,
2                       out queue<Replica_In>: inputs
3                       in queue<uint32>: active_members)
4  {
5    var Replica_In: tmp;
6
7    state Main{
8      encapsulate_inputs(a, b, tmp);
9
10     loop over active_members with it {
11       inputs.push(tmp);
12     }
13   }
14 }
```

Fig. 2. Design pattern for the initiator.

A. Initiator

Even though it does not appear as a standalone concept in the original design of NVP, the initiator is introduced and serves three different purposes:

- First of all, member versions are assumed to run concurrently and work on the same data, hence a suitable mechanism must be employed to properly trigger their execution. Actually, this aspect is considered as part of the NVP execution environment illustrated in [2].
- Secondly, as it will be better explained in Section IV-D, this way of modeling is essential to preserve the timing characteristics of the original process.
- Thirdly, as mentioned before, the initiator plays a key role in maintaining the same interfaces the original process has with the rest of the system.

The initiator is mainly responsible for populating inputs to (existing) member versions, which also triggers their execution. Since member versions are derived from the same initial specification, the alternative algorithms implemented in them work on the same inputs and outputs as defined in the initial specification, which also corresponds to the inputs and outputs specified in the original process. Besides, the initiator also inherits other attributes from the original process, such as its period, offset, and activation conditions, so that the timing behavior will be maintained as well (see Section IV-D).

Fig. 2 sheds more light on the design pattern for the initiator. For demonstration purpose, it is assumed that the original process simply takes two inputs (`a` and `b`), calculates the sum and outputs it. The `inputs` queue (line 2) represents the queue used to exchange information between the initiator and the member versions shown in Fig. 1, in particular it collects inputs for member versions. Since the original process can have various inputs (in terms of both number and data type) whereas queue elements are homogeneous, `Replica_In` is a data structure designated to accommodate heterogeneous inputs and provide a uniform interface between the initiator and member versions. It also offers flexibility for extension. As shown in Fig. 2, inputs (e.g., arguments `a` and `b`) will be first encapsulated into this data structure by means of a standard function (lines 8) before being delivered to member versions through appropriate queue operation (lines 10–12).

```
1  processdef Member_Version(in queue<Replica_In>: x,
2                            out queue<Comp_Vector>: z,
3                            out queue<uint32>: status,
4                            in uint32: id)
5  {
6    var uint32: a;
7    var uint32: b;
8    var uint32: sum; /* An uint32 holds the result here */
9    var Replica_In: tmp_in;
10   var Replica_Out: tmp_out;
11   var Comp_Vector: my_result;
12
13   common{
14     tmp_in = x.pop();
15     unfold_inputs(tmp_in, a, b);
16
17     /* Other common code goes here. */
18   }
19
20   /* User-written FSM goes here without modification. */
21   state Main{
22     sum = a + b;
23   }
24
25   finally{
26     encapsulate_outputs(sum, tmp_out);
27     my_result.mem_res = tmp_out;
28     my_result.mem_id = id;
29
30     z.push(my_result);
31     status.push(id);
32   }
33 }
```

Fig. 3. Design pattern for member versions.

It is worth noting that the concept of *iterator* implemented in the CPAL language together with the `loop over` operator allows to sweep through a collection (be it an array, a queue or a stack) in a consistent way, regardless of the number of elements available in it. `active_members` is a queue that holds the decision from the voter regarding each member version, whether it is considered to be faulty and its execution should be terminated or not. If so, consequently, the initiator will stop supplying inputs for it. In particular, the queue contains *identifiers* of non-terminated member versions.

As we can see, the only variable part of the initiator is the inputs that can be grouped into the `Replica_In` structure and treated as a separate, small *domain-specific* model. The remaining part is quite standard for automation regardless of the original process to be NVP-enhanced.

B. Member Versions

Fig. 3 depicts the design pattern for the process definition corresponding to a member version. A key feature of this pattern is that it encapsulates the user-provided FSM, responsible of implementing the member version logic and symbolized by lines 21–23 of the listing. A member version works on the inputs it receives from the initiator through the input queue. In turn, it generates a comparison vector based on its own state value obtained through the logic implemented in the user-provided FSM and informs the voter about its execution status through a `status` queue (lines 1–4). Parameter `id` is an identifier that can be freely configured by system designers and used to recognize a member version uniquely. As shown in Section IV-D, this is done at processes instantiation.

First of all, inputs encapsulated in the `Replica_In` data structure and held in a queue element should be unpacked (lines 14–15) before they can be referred to by the user-written FSM as it does not work directly on the data structure. This can be achieved by means of the `unfold_inputs` function. Correspondingly, redundant variables are created (lines 6–8). This way of doing aims at maximizing system designers' freedom during the implementation of a member version algorithm while keeping fault-tolerance related code as *independent* as possible so as to minimize interference to the user code. It is also worth noting that the unpacking operation should be performed every time a member version is activated.

A CPAL process is mainly made up of state and transition code. Besides, CPAL offers native support to specify code shared among all states, by means of a `common` and `finally` block. An *execution step* of a CPAL process starts with the evaluation of state transitions which determines the next, target state and then it proceeds with any pre-state common code (the `common` block). Afterwards, it executes the code pertaining to the target state, and, lastly, any post-state common code (the `finally` block). As we can see, the `common` block provides a convenient place to unpack inputs. Since the user-provided FSM may already include some common code, the code for unpacking the input data should stay at the very beginning. As mentioned before, in order to let the user-written FSM remain unaltered, local variables with the same name as the parameters of the original process are created (lines 6–8). It is important to remark that, even though the member logic FSM shown in Fig. 3 is fairly simple (it merely adds the two inputs), the design pattern supports arbitrary FSMs.

Moreover, the `finally` block makes it straightforward for a member version to export its comparison vector and notify its execution status to the voter (lines 25–32). The comparison vector, represented by the `Comp_Vector` data structure, includes values calculated by a member version and relevant for voting (encapsulated in the `Replica_Out` data structure), as well as a field which identifies the producer of the comparison vector. The comparison vector is updated to the voter via a queue shared between all member versions and the voter. Besides, a member version also pushes its `id` to a status queue which informs the voter that the corresponding member version has completed its calculation. As shown in Section IV-D, it will be used to trigger the execution of the voter.

The above way of modeling indicates that the *cross check point* is implicitly set to the end of each execution step, which represents the finest granularity (namely, *state level*) ever achievable. Coarser granularity can be achieved by making the export of the comparison vector conditional, e.g. every multiple number of periods.

C. Voter

The voter performs comparison based on the results produced by the member versions and generates outputs used by other parts (e.g., ext_4, ..., ext_6 in Fig. 1) of the modeled system. Besides, it also takes suitable actions, *terminate* or

```
1  processdef Voter(in queue<Comp_Vector>: v,
2                   in queue<uint32>: status_queue,
3                   out uint32: sum,
4                   out queue<uint32>: alive_members)
5  {
6    var Ballot_Grouping: majority;
7    var queue<Ballot_Grouping>: summary[NUMBER_OF_VERSIONS];
8
9    state Main{
10     Majority_Voting(v, majority, summary);
11
12     unfold_outputs(majority, sum);
13
14     alive_members.clear();
15     loop over v with it{
16       if(comp_ballot(it.current.mem_res, majority.value)){
17         alive_members.push(it.current.mem_id);
18       }
19     }
20
21     v.clear();
22     status_queue.clear();
23   }
24 }
```

Fig. 4. Design pattern for the voter.

continue, for each member version according to the result it provides. More specifically, *continue* indicates that the result provided by a member version is in line with the one returned by the voter, otherwise a member version will be considered faulty and *terminated*. Termination of a member version is achieved by disabling the activation condition of the corresponding process, as shown in Section IV-D. Re-activating the process can possibly be done by an application-dependent re-activation condition.

As depicted in Fig. 4 lines 1–4, the voter works on the comparison vectors and execution status indicators collected from member versions, through the two different queues explained in Section IV-B. In turn, the voter exports the same outputs as specified in the initial requirement. Feedback to member versions is delivered through a separate queue, namely `alive_members`, which includes the `id` of non-terminated member versions determined by the voter.

As remarked in Section II, majority voting (line 10) is used as the decision algorithm in this paper. Like other decision algorithms, it will be made available through library functions. Hence, the exact code is not shown here for clarity. The majority voting algorithm just requires in input the set of the comparison vectors and it returns the majority result (if it exists) and a summary that reports the statistics of the vote.

More specifically, it scans through elements in the queue, keeps record of each occurred ballot and collects statistics, for instance, how many member versions agree on a particular ballot. This information is kept in a separate queue internal to the voter (line 7), whose elements are of type `Ballot_Grouping`, which in turn is made up of two fields: a field of type `Replica_Out` and a counter.

Moreover, along the way, the voter keeps track of and refreshes the ballot with the maximum number of member versions supporting it. This makes it convenient to determine whether or not a majority exists by simply comparing it with the total number of member versions currently running in the system. If majority is reached, the outputs will be

```
1  /* process Original_Process: origin_proc[100ms]
2                               (input1, input2, output1); */
3
4  process Initiator: initiator[100ms]
5                 (input1, input2, input_queue, active_members);
6
7  process Member_Version: m1[]
8                 [member_alive(active_members, id)
9                  and input_queue.not_empty()
10                 and (not exec_complete(status_queue, id))]
11                (input_queue, id, comp_vectors, status_queue);
12
13 process Voter: voter1[]
14                [comp_vectors.not_empty() and
15                 comp_vectors.count() == status_queue.count()]
16                (comp_vectors, status_queue,
17                 output1, active_members);
```

Fig. 5. Design pattern for process instantiation.

updated (line 12) according to the majority result obtained. Otherwise, warning information should be returned to gather further diagnostics and possibly implement fail-safe behavior.

After that, the voter evaluates whether the results produced by a member version match the majority or not (by means of the `comp_ballot` function) and flags member versions for termination accordingly (lines 14–19). This operation can be performed efficiently again with the *iterator* offered in CPAL, with the `it.current` construct returns the current element in the collection. At the end, the queue storing the comparison vectors and the execution status indicators are cleared and reset (lines 21–22), so as to prepare for the next round of execution. As we can see, the voter code is standard as well and works independently from the configuration of the total number of member versions (denoted `NUMBER_OF_VERSIONS`).

D. Validation

Fig. 5 depicts how to construct the NVP module corresponding to an initial specification, from the individual patterns described in previous sections, by means of a simple but yet representative example. The original process (shown commented out at the top of the figure) is just used as a reference for readers. It is a periodic process that works on two input variables and generates one output. For the sake of clarity, Fig. 5 just shows the template for member version instantiation regarding only one member. `id` is the unique identifier to be specified for each member version, which is also used to facilitate the termination of a member version.

Synchronization among different components of the NVP module can be conveniently specified through the *activation conditions* of process instances. More specifically, the execution of a member version (lines 8–10) depends on whether it has been previously terminated by a voter (evaluated via the `member_alive` function), whether there are inputs from the initiator as well as it did not complete its computation for the current round yet (determined by the `exec_complete` function). On the other hand, the voter will be triggered upon the completion of all existing member versions (lines 14–15). As we can see, both the member versions and the voter are modeled as pure *event-triggered* processes, while the initiator can be either periodic, event-triggered, or hybrid depending on the original process.

The correctness of the design patterns has been checked experimentally in the *value* domain by artificially manipulating the result calculated by a member version so that it is different than the one generated by the other two versions. In this case, the voter was always able to produce the right output and identify the faulty member version. On the other hand, since the member versions and the voter are purely event-triggered, they will run after the initiator within the same cycle when activation conditions are fulfilled. Hence, the execution characteristics of the overall NVP module depend on the one specified for the initiator, which is actually taken from the original process. As a result, we could confirm that system timing characteristics are also preserved.

In summary, NVP is modeled in such a general and compact way that it can be flexibly applied to any module of the modeled system without changing its original behavior. In particular this holds true for what concerns, first, its way of interacting with other modules of the same system and, second, the correctness regarding both the value and time domain. These advantages also make the NVP model *composable*, namely it can be applied to more than one module at the same time and system properties are still preserved. Moreover, the same modeling strategy can be easily ported to other fault-tolerant techniques, e.g. recovery blocks. Last but not least, it is possible to automatically generate the NVP template code shown through Figs. 2–5 by code transformation tools, provided that the user-supplied member versions FSMs are available.

V. C Language NVP Implementation

The NVP implementation consists of a C-language [13] library, called `Ft` in the following. The goal of the library is to provide several basic building blocks, which users can customize and blend with their own application code to build a complete fault-tolerant system, bearing a strong likeness with the design patterns presented in Section IV.

The choice of C as implementation language is the result of a tradeoff. Its sheer diffusion makes `Ft` very portable, even to targets with limited memory and processing resources, which may poorly sustain the overhead of more sophisticated runtime systems. Moreover, the C language is well-known to most embedded system programmers, thus smoothing out the learning curve to incorporate fault-tolerance code in their software. Last, but not least, open-source toolchains for the C language are readily available and relatively simple to configure, build, and integrate with a Real-Time Operating System (RTOS) [14]. These advantages offset the disadvantage that C is a relatively low-level language and lacks some features important for a proper, type-independent implementation of `Ft`, most notably C++ *templates* or ADA *generic units*.

A. Lightweight C-Language Templates and Data Types

C++ language *templates* allow programmers to define classes and methods in which one or more data types and other elements are replaced by placeholders and left unspecified. These entities cannot be used directly—they cannot even be compiled due to lack of data type information, for instance—but can be *instantiated* one or more times. Placeholders are filled upon instantiation, thus providing the missing information and making code generation possible. In this way, the resulting classes and methods can be tailored to specific data types without source code duplication.

Several aspects of `Ft` benefit from this feature. For instance, the implementation of the voter algorithm modeled in Section IV-C should be written in a way that is independent on the data type used for votes in a specific application. Another example is the encapsulation of a member version, presented in Section IV-B, within an RTOS tasks to facilitate its concurrent execution. Also in this case the encapsulation code should be developed only once, even though the data types used for member version's operands and return value may vary.

In the simplest cases, this goal can be accomplished by multi-layer macro expansion. For instance, the following fragment of code defines a template for the data type `FtBallot_`*`type`*, where *`type`* is a placeholder for another data type. The data type being defined represents a ballot and is a structure with two members: an identifier of the agent that cast the vote, and the vote itself. Although the identifier can invariably be represented as an integer, the vote data type depends on the application. It can be specified by instantiating the template with a specific *`type`*. As an example of invocation, `FT_BALLOT_TYPEDEF(uint32_t)` gives rise to the definition of the `FtBallot_uint32_t` data type containing a `uint32_t` vote.

```
#define FT_BALLOT_TNAME(btype)                  \
    FT_BALLOT_TNAME_X(btype)

#define FT_BALLOT_TNAME_X(btype)                \
    FtBallot_ ## btype

#define FT_BALLOT_TYPEDEF(btype)                \
    typedef struct {                            \
        int id;                                 \
        btype vote;                             \
    } FT_BALLOT_TNAME(btype)
```
(1)

As a language-related technical note, it should be remarked that the multiple layers of macro invocation shown in (1) are indeed necessary. They force the expansion of argument `btype` inhibited by the **##** token concatenation operator [13, Clause 6.10.3.3]. Unfortunately, this method is not adequate—mainly because of code readability issues—when the macro body becomes more complex as it happens, for instance, when a whole function contains placeholders. In those cases, a different approach has been used, involving the inclusion of a source module within another, a possibility explicitly allowed by the standard [13, Clause 5.1.1.1].

It should be noted that `Ft` uses placeholders also for cases in which a regular function argument could apparently accomplish the same goal—for instance, the number of member versions. The main advantage of placeholders is that their value is known at *compile time* rather than at *run time*. This helps to avoid dynamic memory allocation of working storage local to the function, a C-language feature that requires extensive use of pointers and is prone to state- and time-dependent failures, possibly involving multiple tasks in the

TABLE I
MAIN DATA TYPES

Type	Purpose
`FtChannelKind_t`	Kind of channel
`FtChannel_t`	Generic channel
`FtTaskFunction_t`	Generic task function
`FtTaskArg_t`	Task function arguments
`FtTask_t`	Task
`FtTimeout_ms_t`	Timeout (in ms)
`FtCollectorFrontend_t`	Collector frontend function
`FtStatus_t`	Status code
`FtBallot_`*`btype`*	Single ballot, def. as in (1)
`FtAllBallots_`*`nball_btype`*	Array of ballots
`FtBallotGroupingEl_`*`btype`*	Group of analogous ballots
`FtBallotGrouping_`*`nball_btype`*	Overall ballot statistics

system. For both reasons, it is often not used in code whose goal is to be highly reliable and is forbidden in CPAL.

Table I summarizes the data types defined and used by `Ft`. Basic data types, listed at the top of the table, have a fixed definition, that is, a definition independent from the data types used, for instance, by voters and member functions. Among them, the most complex is `FtChannel_t`. It provides an OS and network-independent streaming communication channel among `Ft`-managed tasks, making their deployment straightforward even on a distributed system. The data types listed at the bottom of Table I are instead defined on top of user-defined data type placeholders through the template-based mechanism previously. In the table, placeholders that become part of the derived data type name are typeset in italics. More specifically, *`btype`* represents the votes data type, while *`nball`* represents the number of member versions.

B. Member Version Encapsulation and Execution

Within `Ft`, at the lowest level of abstraction *member versions* are implemented as subroutines with two arguments. The first argument (passed by value) gathers all input data, while the second (passed by reference) holds results—that is, the vote cast by the member version. The reason of grouping all input data into a single argument, and likewise for results, is to avoid functions with a variable number of arguments that would make encapsulation harder to implement. Starting from these user-defined functions, `Ft` provides two levels of abstractions towards a full-fledged NVP system. The first level abstracts away from member function deployment and user-defined data types, while the second achieves underlying OS independence. They are kept separate to improve modularity.

1) Task function encapsulation: The first abstraction encapsulates a member version function within a *task function*. This function has a fixed, uniform prototype, determined by the `FtTaskFunction_t` data type listed in Table I and is suitable to be further encapsulated within a thread that can then be scheduled for execution anywhere in the system. More specifically, a task function has a single argument of type `FtTaskArg_t`. The members of this structure are a pointer to the task function itself (whose role will be better detailed in the following) and two channels of type `FtChannel_t`, `in` and `out`. They are used to read input arguments and deliver the ballot to the voter, respectively.

The use of input and output channels instead of arguments that directly hold (or point to) memory-resident data structures, like it happened for member version functions, enables a very flexible deployment of task functions in a distributed system. In fact, channels support data transfer between functions even across different hosts. What is more, using a message passing communication model instead of shared memory, alleviates synchronization issues regardless of the task scheduling policy.

As an additional benefit, the use of channels makes the task function signature completely independent of user-defined data types. On the other hand, including a pointer to the task function itself in its arguments allows an argument of the same data type to be used for the trampoline function to be discussed next. Encapsulation is performed by macro invocation, as `FT_VERSION_TASK(name, id, f, intype, btype)`, where `f` is the member function to be encapsulated, `intype` and `btype` are the data types of its `in` and `out` arguments, respectively, and `name` is the name of the task function that the macro will define. Finally, `id` is an integer identifier that will be used to uniquely tag its ballots and corresponds to the `id` field of the ballot data structure mentioned in (1).

2) Task encapsulation: The section abstraction takes a task function just described and instantiates a task to execute it, independently from the underlying operating system and its multitasking API. The portability of this abstraction and of `Ft` as a whole has been confirmed by providing support for operating systems based on the *POSIX threads* API [15]—for instance, GNU/Linux and RTEMS [16]—as well as the proprietary API of FreeRTOS [17].

The function `FtTaskCreate(f, in, out, task)` takes a task function `f` to be encapsulated and its `in` and `out` channels as inputs. It instantiates a task to execute `f` on the same node it is invoked on and fills a `task` identifier of type `FtTask_t` when successful. The main issue to be addressed within `FtTaskCreate`, to achieve operating system independence, is that different multitasking APIs stipulate different signatures for task entry points. For instance, in the POSIX threads API the thread entry point is a function that returns a `void *` that represents the task termination status, whereas FreeRTOS does not directly support this feature. Moreover, other OS may very well specify dissimilar signatures.

As a consequence, the OS-dependent part of the implementation of `FtTaskCreate` must interpose a *trampoline* between the OS-dependent task entry point and the task function itself, an OS-independent `FtTaskFunction_t`. Incorporating a pointer to the task function itself within the `FtTaskArg_t` allows the same pointer to this structure to be used as an argument to both the trampoline and task functions. In this way, the trampoline can invoke the task function through it, and forward it to the task function with minimal overhead and very light requirements on the underlying OS—virtually all of them support passing a pointer as argument of a task. At the same time, this technique closely parallels the modeling method described in Section IV-B.

C. Majority Voting

The voting mechanism, here illustrated by describing how majority voting works, is articulated as a cascade of two software modules, to be discussed in the order they process ballots. Like for member version encapsulation, discussed in Section V-B, the reason is to ensure a proper separation of duties among them and make `Ft` more modular and flexible. Both modules are implemented by means of template instantiation, as described in Section V-A.

1) Collector: The collector is responsible of repeatedly invoking a *front end* to collect ballots and move them into a local, memory-resident data structure. To further enhance modularity, the front end is a function that is responsible of retrieving ballots one by one and make them available to the collector, along with status information about the outcome of the operation. In the scheme discussed in this paper, the front end retrieves the ballot from the `FtChannel_t` used by task functions as output (see Section V-B). Receive timeouts reported by channels—indicating that a member version was unable to provide its ballot in time—are also propagated to the collector. As is done in other cases, in order to reduce `Ft` development effort and enhance code quality, specific collectors are generated from generic templates, having the number of ballots to collect and their data type as placeholders.

2) Voter: The voter has an input argument of type `FtAllBallots_nball_btype`. It is an array of `FtBallot_btype` defined as shown in (1) and contains ballots from the collector. Its output contains vote statistics and is of type `FtBallotGrouping_nball_btype`. Individual elements, of type `FtBallotGroupingEl_btype`, summarize input ballots that contain homogeneous votes. Accordingly, they contain a vote count and the exemplar vote value. Voters themselves are instantiated from voter templates that have the expected number of ballots and the vote data type as placeholders.

It should be noted that the interface between collector and voter depends neither on the ballot source (handled by the collector front end), nor on the voting algorithm (internal to the voter). It is therefore possible to reprogram the system to use a different voting algorithm in exactly the same way as it is done in the modeling phase—that is, by simply replacing the voter module. Similarly, collecting ballots from a different source merely requires the replacement of the collector front end, without affecting voter's operation and correctness. Recompilation of unaffected modules is not needed, either.

At present, there is no feedback from the voter to member versions in `Ft`. Faulty versions are flagged for higher-level reporting, but they are also kept for the whole lifetime of the system, relying on the fact that invalid votes are consistently discarded by the voter and channel timeouts detect missing votes. The kind of feedback outlined in Section IV-C can easily be added though, by post-processing and correlating voter's inputs and outputs. In our running example, correlation can detect votes that disagree with the majority. Then, the member versions that cast them can be tracked by means of the unique identifier tagged to each vote (see Section V-B).

VI. Conclusion

In this work, we presented a framework for NVP modeling which is not only faithful to the original concept of NVP and but also offers a generic solution thanks to the design patterns it defines. Most importantly, it keeps the fault-tolerant mechanisms independent from the logic of the application. Such patterns enable system designers to explore the use of fault-tolerant mechanisms in the early design phases of a system without being overwhelmed by implementation details, but just focusing on the application-dependent functional logic. Building on this study, our ongoing work aims at a complete automation of the code instrumentation for the NVP as well as other prominent error-detection and error-correction mechanisms. We are also developing a software fault-injection framework to help system designers validate in a quantified manner the effectiveness of the fault-tolerant mechanisms, and thus the dependability of the system.

References

[1] B. Randell and J. Xu, "The evolution of the recovery block concept," in *Software Fault Tolerance*, M. R. Lyu, Ed. John Wiley & Sons, Inc., 1995, pp. 1–22.

[2] A. Avižienis, "The methodology of N-version programming," in *Software Fault Tolerance*, M. R. Lyu, Ed. John Wiley & Sons, Inc., 1995, pp. 23–46.

[3] B. Baudry and M. Monperrus, "The multiple facets of software diversity: Recent developments in year 2000 and beyond," *ACM Computing Surveys*, vol. 48, no. 1, pp. 16:1–16:26, Sep. 2015.

[4] N. Navet and L. Fejoz. (2017, Jan.) The CPAL programming language, version 1.06. [Online]. Available: https://www.designcps.com/wp-content/uploads/cpal-intro.pdf

[5] R. T. Wang, "A dependent model for fault tolerant software systems during debugging," *IEEE Transactions on Reliability*, vol. 61, no. 2, pp. 504–515, Jun. 2012.

[6] Hystrix. Latency and fault tolerance for distributed systems. [Online]. Available: https://github.com/Netflix/Hystrix/

[7] P. Hosek and C. Cadar, "VARAN the unbelievable: An efficient N-version execution framework," in *Proc. 12th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2015, pp. 339–353.

[8] G. Latif-Shabgahi, J. M. Bass, and S. Bennett, "A taxonomy for software voting algorithms used in safety-critical systems," *IEEE Transactions on Reliability*, vol. 53, no. 3, pp. 319–328, Sep. 2004.

[9] M. Rezaee, Y. Sedaghat, and M. Khosravi-Farmad, "A confidence-based software voter for safety-critical systems," in *Proc. 12th IEEE International Conference on Dependable, Autonomic and Secure Computing*, Aug. 2014, pp. 196–201.

[10] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud, "The synchronous data flow programming language LUSTRE," *Proceedings of the IEEE*, vol. 79, no. 9, pp. 1305–1320, Sep. 1991.

[11] P. L. Guernic, T. Gautier, M. L. Borgne, and C. L. Maire, "Programming real-time applications with SIGNAL," *Proceedings of the IEEE*, vol. 79, no. 9, pp. 1321–1336, Sep. 1991.

[12] F. Boussinot and R. de Simone, "The ESTEREL language," *Proceedings of the IEEE*, vol. 79, no. 9, pp. 1293–1304, Sep. 1991.

[13] *ISO/IEC 9899, Programming Languages — C*, 2nd ed., International Organization for Standardization and International Electrotechnical Commission, Dec. 1999.

[14] I. Cibrario Bertolotti, "RTOS support in C-language toolchains," in *Proc. 18th IEEE International Conference on Industrial Technology (ICIT)*, Mar. 2017, pp. 1328–1333.

[15] *ISO/IEC/IEEE 9945, Information Technology — Portable Operating System Interface (POSIX®) Base Specifications, Issue 7*, IEEE and The Open Group, Sep. 2009.

[16] On-line Applications Research Corp. (2011, Dec.) RTEMS documentation. [Online]. Available: http://www.rtems.com/

[17] R. Barry, *Using the FreeRTOS Real Time Kernel — Standard Edition*, 1st ed. Raleigh, North Carolina: Lulu Press, 2010.