Deterministic Parallel Programming for Railway Applications

Oscar Medina Duarte
Thales Austria GmbH
Handelskai 92, 1200 Vienna, Austria
Email: oscar.medina@thalesgroup.com

Peter Tummeltshammer
Thales Austria GmbH
Handelskai 92, 1200 Vienna, Austria
Email: peter.tummeltshammer@thalesgroup.com

***Abstract*—Transport Automation systems rely on stringent reliability and availability specifications which are met by implementing replication techniques. At the same time, the microprocessor market is transitioning to multi-core technology which comes with the promise of increased computing performance, energy efficiency and alike. Parallel programming is a common technique used to exploit the new capabilities; however, this is also associated with non-determinism. Determinism is a requirement for replication of synchronous systems, which is why parallel programming is still a challenge for safety-critical applications.**

This paper presents an architectural description of a well-established railway-control platform subject to dependability certifications, its programming restrictions, and some of the aspects associated with the introduction of parallel programming into that environment. A formal semantics framework of parallel programs is extended to show how the guarantees provided by Deterministic MultiThreading (DMT) techniques apply to replica determinism. Finally, the paper presents an outlook on the applicability of DMT techniques in railway domain applications.

***Keywords*—Replica determinism, Railway Safety, Parallel Programming, Fault tolerance**

I. Introduction

Transport Automation (TA) systems share several architectural characteristics derived from the fact that they are subject to certifications with respect to their functional requirements and their dependability[1]. Software replication is a common technique used in TA systems in order to achieve the levels of dependability required by the normative applicable to a specific industry (e.g., CENELEC in the railway control industry). Replicating a system function requires that each replica exhibits the same observable output at a given time for the same input [2]; this is referred to as *replica determinism* [2]. This assumption is straight forward in the sequential programming model where there only exists one possible sequence of instructions and memory accesses (for the same input). Parallel programming on the other hand, does not necessarily provide any guarantees with respect to the order in which the instructions of a program are executed [3]. *Deterministic MutliThreading* (DMT) is a class of parallel programs which solve the non-determinism inherent to mutlithreaded parallel programming by applying memory access scheduling techniques [4], [5], [6]. This paper reviews the implications of introducing parallel programming into a well-established Transport Automation Service Platform (*TAS Platform*) [7], [8] and discusses the suitability of DMT techniques for this purpose.

Aspects relevant to the certification, architecture and programming interface of a safety-critical TA platform in the railway domain are revealed in Section II. Further, Section III provides a discussion about the determinism of parallel programs, and the potential of DMT in replicated software systems. Section IV gives an insight on the prospects of parallel technology in the railway industry. Section V concludes by arguing in favour of the application of DMT in safety-critical applications.

[1]Dependability is defined as the "ability to avoid service failures that are more frequent and more severe than acceptable" [1].

II. Railway-control platform

The TAS Platform is a well-established base technology for safety-critical railway applications such as *electronic railway interlocking*, *axle counters* and *automatic train control systems* (e.g., the European Train Control System or ETCS) among others [7], [8]. The main goal of this platform is to provide a generic safety middleware with fault-tolerant and real-time capabilities that supports the fulfilment of the overall RAMS requirements (described Section II-A), while maintaining a strict separation between the safety related functions and the rest of the system. Figure 1 presents the layered design approach of the TAS Platform. It can be observed that the Safety-critical applications are constructed on top of a safety middleware, which isolates safety-related characteristics of the platform like replica-determinate communication, fault detection, application recovery, re-integration and restriction of un-safe calls to the operating system.

This separation has the benefit of masking away the details of replication, real-time and safety-related features towards the software developer. The application programming interface (described in Section II-C) is based on the POISX standard, which is familiar to UNIX/Linux/C/C++ programmers and it also improves portability.

A. Dependability requirements in railway automation

Depending on the country and region of operation, railway control software must comply with a variety of local and international standards. In the case of Europe, railway control systems are subject to certification according to the CENELEC standards. The European Norm 50126 [9] specifies

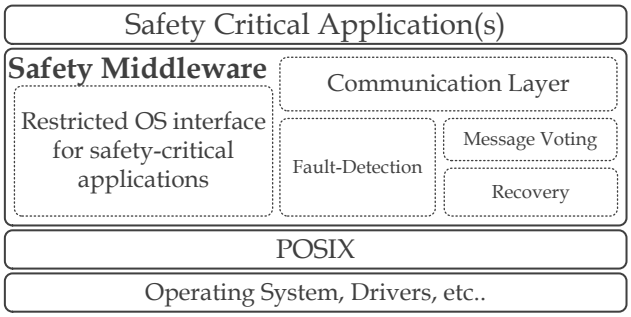

Fig. 1. Layered architecture of the *TAS Platform* [8]: Safety-critical applications gain access to system resources through a middleware layer, which transparently provides the services required to operate within the restrictions of a safety case for the application.

the basic Reliability, Availability, Maintainability, and Safety (RAMS) requirements applicable to railway control systems, the EN 50128 [10] and EN 50129 [11] standards specify the requirements for the development of software and electronic devices in the railway control domain. During the risk analysis[2] phase of a system life-cycle, the hazards of a system are identified and analysed, resulting in tolerable hazard rates (THR) for each hazard [11]. For those systems whose failure may result in significant human, economic or environmental damage, the THR is that of SIL4 [13] which is equivalent to a maximum of 10^{-8} dangerous failures per hour (i.e., less than a *single* dangerous failure every $11,415$ years).

Applications in the railway domain differ from applications in other domains (e.g., avionics) by the fact that in several scenarios, it is possible to safely stop the delivery of a service in the event of a detected anomaly [13]. For example, an axle counter application monitors the occupancy of a railway track section in order to maintain a safe distance between trains [14]. If a segment is occupied, other trains are not allowed to enter or to cross that section. To ensure safety, the equipment must continually monitor its own condition during operation (health monitoring). When a fault is detected, the system will be switched into a defined safe state (safety reaction), which in this case would mean to indicate that the track section is occupied, and that the system is in a faulty state [14], [15]. This is an example of a safety reaction that ensures a safe operation but degrades the availability of the infrastructure (by not allowing trains to traverse the faulty section).

A system that is able to deliver "continuous operation in the presence of faults," [1] is known as a *fault-tolerant* system [1]. Fault-tolerant systems compare the various outputs of the replicated functions: If all outputs are the same, the system is considered to be fault-free. Replication is used not only to detect faults, but like in the case of Triple Modular Redundancy (TMR), it is also used to achieve higher availability through recovery and re-integration of failed nodes [13].

[2]The IEC/EN 61508 international standard uses a formulation of risk assessment on the basis of the *average probability of failure per hour* [12] and defines four Safety Integrity Levels (SIL1 to SIL4) that are used as a classification for the tolerated failure rates of safety-critical components.

An important aspect of the correctness of a TA system is the timeliness of its behaviour. Each computation and each reaction has to be performed within a defined time interval or deadline. The term *real-time* is used to denote these types of systems. A *real-time computer system* is defined as "a computer system in which the correctness of the system behaviour depends not only on the logical results of the computations, but also on the physical time when these results are produced" [2]. As a consequence, time behaviour is supervised by the system in order to make sure that it is compliant with the requirements established during the design phase of the application and ensure a prompt reaction when faults are detected [13].

Our research aims for the introduction of parallel programming models into an existing TA platform that provides a fully transparent replication layer to the software applications. In the following sections we explain some of the requirements for transparency in replication and give examples where concurrent programming falls short at complying with them.

B. Replication

Replication is the association of N independent modules with identical functionality. The TAS Platform is built upon the K-out-of-N replication architecture. A system formed by N modules is said to be K-out-of-N redundant if the system is able to be operational if and only if at least K out of its N replicas are non-faulty [16], [17]. The assumption is that when each of the non-faulty modules receives the same input, each of the non-faulty modules shall produce the same output. In order to verify this, the replicas exchange messages through a redundant network (as depicted in Figure 2) containing their respective computed values. A consistency criterion or *voting-function* is applied in order to detect faults and to reach an agreement between the different replicas (see Figure 3). The assumption also requires that the modules are synchronized because the input for each of the voted messages shall correspond to the same computation, that is, the messages shall be consumed in an order consistent to all of the replicas.

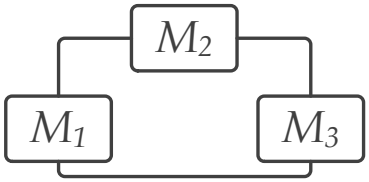

Fig. 2. Triple Modular Redundancy communication: The modules exchange messages with each other in order to detect errors, maintain a global time and reach a consistent agreement of states among the replicated modules (i.e., *interactive consistency* [18]).

The middleware layer of the TAS Platform provides transparent access to the required synchronization, communication, fault detection (voting) and health-monitoring services that the applications need in order to support the following K-out-of-N redundancy configurations [8]:

1-out-of-1 or non-redundant configuration. In this configuration, health monitoring and software diversity are com-

mon measures provided by the TAS Platform in order to ensure safety [13].

2-out-of-2 configuration. The voting function in this configuration has the capability to detect that a fault has occurred, that is, if the two messages are not identical, then a safety reaction is activated. This mode of redundancy allows for an application to implement a safe failure mode like *fail-stop* [19]: Upon the detection of a failure, all replicas stop communicating.

2-out-of-3 configuration, also referred to as Triple Modular Redundancy (TMR). The voting function in this configuration, has the ability to mask a single failure, determine the faulty module and continue to operate without service interruption (see Figure 3). This redundancy configuration is aimed at increasing the availability of a system.

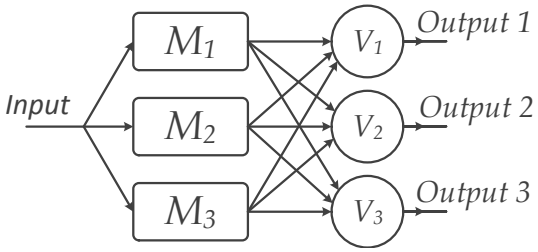

Fig. 3. TMR System with distributed voters: A distributed majority voting function is used in order to unify the states of the different processors [18] [20] and produce a single correct output (a comprehensive survey of different voting algorithms can be found in [21]).

A voting mechanism in a real-time system requires not only consistent information but also timely information. S. Poledna [22] provides a definition of *replica determinism* to describe the behaviour of non-faulty replicas as "a correspondence of server outputs and/or service state changes under the assumption that all servers within a group are starting at the same initial state, executing the same service requests within a given time interval".

C. TAS Platform Programming Model

The programming model of the TAS Platform aims at allowing an application developer to write software with main focus on the functional aspects of the program. That is to say that those aspects related to replication, synchronization, communication, fault detection and recovery are mostly transparent from the programmers' point of view. In this context, transparency refers to the separation between an application (functional part) and the services provided by the middleware (non-functional part).

Using the TAS Platform programming model, allows the middleware to be configurable with respect to the hardware redundancy architecture without having the need to modify the source code of an application. For example, it is possible to configure a system to use any of the redundancy configurations described in Section II-B without modifying the application. The programming model is accompanied by a set of programming restrictions with the purpose of ensuring that the requirements and assumptions of the underlying platform with respect to safety (*Safety Case*) and replication (*replica determinism*) are not violated. The strict separation between the functionality of an application and safety related services has the benefits of reducing the certification efforts and improving the overall safety of an application.

In order to satisfy the time requirements and to guarantee replica determinism, the middleware of the TAS Platform provides a scheduled parallel tasking model. In this model, tasks belong to a set of tasks (group) and can communicate locally with other tasks within their group. The tasks are scheduled by a distributed, replica deterministic algorithm that selects only one task from a task group to be runnable at a time. In order to ensure replica determinism, the application shall be restrained from using asynchronous functions (e.g., `signal()`), blocking functions (e.g., `wait()`), non-deterministic functions (e.g., `rand()`) or any other functions that are not replica deterministic (e.g., `gettimeofday()`) including the dynamic creation and destruction of threads within the control flow of a task. Tasks that want to communicate with tasks belonging to other groups, have to do it through the *communication and synchronization layer* of the middleware, which implies that the data is globalized and voted over all hardware replica. This tasking model provides coordination between concurrent tasks to ensure replica determinism, however, traditional parallel programming languages and models cannot be transparently introduced without affecting the safety assumptions, programming restrictions and generally the support from the middleware. In the following section, we will elaborate on the implications of parallel programming with respect to determinism.

III. Determinism of parallel programs

Sequential programming is well understood and proper techniques and programming models exist that will guarantee consistency and repeatability of computations even under multi-programmed environments (e.g., running under a multi-process operating system). Also in the safety-critical domain, these types of programs have been widely used and their verification strategies are well-established in the industry, mainly by means of testing.

Parallel Programming on the other hand, relies on the notion of *concurrency* which refers to the fact that the order in which the instructions of simultaneously executing threads is not predetermined by the program [3], but instead, by the underlying scheduler. Dynamic scheduling decisions may result in temporal instruction interleavings that lead to unwanted behaviours. It is difficult for a developer to predict all possible ways in which her code could be executed by the computer [23].

A. Sources of Non-Determinism

Sources of replica non-determinism can appear at every layer of an architecture, for example, still under the assumption that a replicated group uses exactly the same type of hardware, slight variations of clock speed may occur, resulting in the various internal clocks to drift apart and possibly

causing *inconsistent ordering* of the messages or *timeouts*. At the operating system level, processing of asynchronous events under a (preemptive) *dynamic scheduler*, may cause different execution sequences (*interleavings*) of operations that can result in timeouts, inconsistent ordering of events, and/or *consistent comparison problems* [24]. Specific to the application layer, sources for replica non-determinism may appear if the application uses local information or other not globally available data sources, like random number generators; uncoordinated local clocks; *non-deterministic program constructs* or any combination of these [22].

Consider an application code like the one in Example 1.

Example 1. Parallel implementation of the *dot product* [3] between two vectors $A = \{a_1, a_2, \ldots, a_n\}$ and $B = \{b_1, b_2, \ldots, b_n\}$. The dot product is defined as $A \cdot B = \sum_{i=1}^{n} a_i b_i$. A natural way to express this problem in OpenMP would be:

```
...
   float result = 0.0f;
#pragma omp parallel for reduction(+:result)
   for (size_t i = 0; i < n; i++){
      result += a[i] * b[i];
   }
...
```

OpenMP will generate a private copy of the *reduction variable* (i.e., **`result`**) on each of the spawned tasks, and will assign regions of the input vectors to each task. The original copy of **`result`** is then updated with the values of the private copies using the specified associative and commutative operator.

Although not immediately evident, the code in Example 1 is not replica deterministic, due to the fact that in OpenMP, the order in which the values are aggregated is unspecified [26].

Finite-precision arithmetic operators are *approximately associative* [24], comparing such results in a distributed voter like the one in Figure 3, leads to a *consistent comparison problem*, which arises whenever a compared quantity is the result of inexact arithmetic [24]. A sequential version of the previous example is replica deterministic, because the order of the computations is always the same, however, its parallel implementation is not replica deterministic since we cannot predict the order in which each of the replicas will apply the reduction operation to the intermediate results.

B. Scheduling of Parallel Programs

This section shows how replica determinism relates to parallel programs by discussing the semantics of instruction interleavings. As we have seen in Example 1, the variability of the order in which the instructions of the threads are executed, may produce different results for the same program with the same input. The parallel semantics of a program can be defined as the set of all possible executions under a given architecture [27]. A program P is defined as an ordered set of communication (i.e., access to shared memory) and synchronization (i.e., locks, barriers, etc...) instructions. This paper uses a notion of *schedule* as the ordered sequence of executed communication and synchronization instructions of a program. For intuition, the order of the executed instructions, is the order as observed by the memory. This definition is inspired by the formal definition of an *execution* in [27], with the difference that we also consider the duration of each instruction in the model.

Two schedules are considered to be *singleton equivalent* if their outputs are the same, for the same input and the order of their instructions is the same [27]. Singleton equivalence is sufficient but not necessary to ensure replica determinism. Replica determinism can be ensured as long as the sequence of inputs and outputs observed within a time interval is the same for a given voting mechanism. Let the ordered sequence of instructions $I = (i_0, i_1, \ldots, i_n)$ be a *schedule* where every instruction $i_k \in I$ is atomic and every pair of distinct instructions holds a happens-before relationship with respect to the memory access. Individual instructions are described by a 4-tuple consisting of an instruction mnemonic (OP), a set of parameters ($val*$), duration of the instruction execution according to a given clock (len) and an identifier to the thread (TID) that executes the instruction[4] e.g., $i_h = (OP, val*, len, TID)$. Instructions can be synchronization operations (i.e., they do not directly interact with memory but they have an effect on the order of the program e.g., *lock/unlock* operations) or memory access operations (i.e., operations which directly read or write memory) [28].

Let $sched(P)$ be the set of all singleton schedules of a program P for a given input.

$$sched(P) = \bigcup_{\forall I \in P} \{I\} \tag{1}$$

Note that the cardinality $|sched(P)|$ is asymptotically exponential to the number of threads in P [5]. If P is a sequential program then $|sched(P)| = 1$ (i.e., one single schedule). If P is a concurrent program then $|sched(P)| \geq 1$.

A multithreaded program that has *one* single schedule ($|sched(P)| = 1$), is defined as a *Deterministic MultiThreaded* (DMT) program [4]. If a program P is the result of concatenating a series of independent tasks τ_r in a sequence, the number of schedules $|sched(P)|$ is equal to the product of the number of schedules of its composing tasks as described in (2).

$$|sched(P)| = \prod_{r=0}^{n} |sched(\tau_r)| \tag{2}$$

For example, the TAS Platform allows task groups to execute concurrently with respect to each other. Tasks within the same group of tasks are scheduled in a sequential (and deterministic) order. In accordance to (2), as long as each task in the schedule is deterministic (regardless of the number of threads involved per task), the overall schedule will also be deterministic.

[3] We reproduce a typical OpenMP programming example from [25] in order to illustrate a source of the *consistent comparison problem* [24] in the context of parallel programming.

[4] A similar definition is given in [27] which does not consider the duration of a given instruction

J. Yang et al. [5] make the observation that many programs rely on a larger number of schedules to efficiently solve a problem and this does not necessarily sacrifice correctness. *Stable MultiThreading* (StableMT) [5] is a class of multithreaded programs in which the number of schedules is reduced by allowing some parts of the code to be non-deterministic (in the singleton sense) with the purpose of enhancing performance. StableMT does not necessarily contradict replica determinism.

Two schedules I_1 and I_2 are said to be *replica-deterministic equivalent* if and only if there is a one-to-one mapping between their inputs and outputs (i.e., *dataflow equivalent* [27]) and its dataflow is observed by a voting mechanism within the same time interval. Replica-deterministic equivalence is necessary in order to construct replicated parallel applications.

We can envision a parallel *Fork-Join* computational model based on DMT or stability to be suitable for the TAS Platform because the execution time of tasks is bounded and the execution policy of task groups provides a similar structure.

Parallel languages like OpenMP [26], Cilk [29] and Cilk*Plus* [30] provide an easy access to the fork-join computation model in C but they are limited at providing determinism by default [31]. DMT and StableMT approaches can help to solve the problem of non replica determinism in distributed replicated applications.

C. Making parallel programs deterministic

We have seen in Section III-A that program behaviour can be non-deterministic depending on how the instructions interleave. As described in Section III-B, DMT is a property of multithreaded programs which guarantees that threads' accesses to shared memory are interleaved in deterministic order repeatable from execution-to-execution of the same program with the same input [4], which provides a determinism equivalent to that of sequential programs [4]. Olszewski et al. [4] defines two styles of determinism in multithreaded programs: *strong determinism* where the order in which instructions access shared memory is deterministic and *weak determinism* which ensures that lock acquisitions occur always in the same order. Programs that are data-race free can be guaranteed to be deterministic using weak determinism. [4], on the other hand, programs with data-races can only be guaranteed to be deterministic if strong determinism is used.

Kendo [4] is one of the first software only DMT implementations which uses a turn-based approach where each thread in a program is only allowed to make shared memory operations while in possession of a token. The turn to own the token is passed from thread to thread in a deterministic order based upon a deterministic logical clock constructed by using a deterministic performance counter [4]. A turn can only be possessed one thread at a time, and only threads possessing the token are allowed to commit memory changes. This technique, while deterministic, incurs high performance costs because all the memory writing operations would be effectively serialized [4]. Threads in load imbalanced programs may have a long waiting time before they get a turn to write [32]. Additionally, Kendo can only guarantee determinism for data-race-free programs i.e., weak determinism [4]. In order to provide strong determinism, systems like Grace [33] or DThreads [6] execute threads in isolation. Each created thread has its own memory space and memory writes are committed into a local memory address. Memory communication occurs at global synchronisation points (i.e., thread creation/termination, mutex lock/unlock, condition variables, barriers and signals [6]) where a deterministic algorithm merges the isolated modifications of each thread into a shared memory. Parallelism is increased by reducing the effects of load imbalance, however, the fact that all threads have to wait at the same global barrier also wastes time that could otherwise be used for computation.

RFDet [28] introduced a relaxed memory model called Deterministic Lazy Release Consistency (DLRC) which relies on the property that threads can only see a memory modification from another thread "if (and only if) the change was made before the currently executing instruction" [28]. This system also isolates the threads' access to memory but it requires no global barriers.

Consequence [34] also proposes a barrier-free DMT implementation but it adopts a kernel-based version controlled memory model [35]. Memory updates can only be seen by a reading thread after a writing thread has *commited* its local memory to the shared memory and the reading thread has requested a memory *update*.

Parrot [32] is an implementation of StableMT. This is built upon the principle of reducing the number of schedules, instead of aiming at solving non-determinism in a concurrent application, it focuses on giving the developer the ability to enable and disable determinism within the code.

The above mentioned systems have the attractive quality that their application programming interfaces are POSIX and can be adopted by simply re-compiling and/or linking an existing program with the respective DMT library. Despite the fact that these systems are not production ready, they provide a good basis for creating deterministic parallel programs in the railway control domain.

IV. Parallelism in Railway Applications

This section discusses the opportunities that parallel programming brings at improving an already well-established software platform. An outlook is given of the applicability of DMT in a safety-and-replication middleware and a taxonomy of railway applications is described which uncovers the aspects that make one application more or less suitable for parallelization using DMT.

Many applications running in the field such as axle counters, point controllers and signal controllers are strongly I/O driven, which means the main source of delay stems from communication and not computation. These applications show little effect to parallelisation attempts, as they are inherently serialised.

On the other hand, there are applications dealing with computationally intensive tasks whose time granularity has the potential to significantly improve the safety or the overall

experience of the users (the users can be railway operators, technicians, passengers or other control systems).

One major driver application for this is future interlocking based on the European Train Control System (ETCS [36], [14]). ETCS is an automated train protection system with particular focus on high speed trains, which was founded by the European Union during the 1990s. The overall goal is to technically harmonize the railway traffic throughout Europe allowing trains to pass borders without having to deal with incompatibilities between the countries. ETCS currently supports three Levels [14]:

ETCS Level 1 works as an advanced intermittent automatic train protection (ATP) system with on-board signalling and can be used as an overlay system with existing signalling systems. Train control information is transmitted by controlled transponders, which get their information from the traditional signalling system.

ETCS Level 2 works as a continuous ATP system in which the train control data is transmitted by radio communication (the current standard: GSM-R). Transponders are used as reference points for the on-board train location system. From the train location data a radio block centre (RBC) calculates movement authorities and transmits them to the trains. Fixed block sections and conventional track clear detection systems still remain.

ETCS Level 3 finally introduces train-borne checking of train integrity which eliminates the need for fixed block sections for line clear detection. With the radio-based train separation (see Figure 4), traditional fixed block sections can be replaced by virtual or moving block.

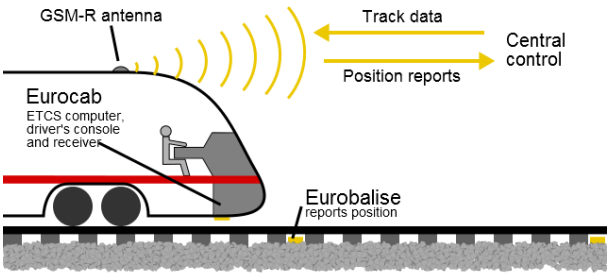

Fig. 4. ETCS Level 3 schematic [37]

The dynamic spacing introduced in ETCS Level 3 will largely increase the computational effort on interlocking, as granting movement authority to a train then depends on dynamic properties of the railway network such as train speed and braking curves, environmental conditions and potentially non-safety related properties such as time tables and power consumption.

The fork-join programming model has been shown to be analysable for real-time applications [38], [39], which makes it a good match to parallelize calculations like the ones mentioned above. Also, as briefly discussed in sections III-B and III-C the fork-join model offers a model suitable for usage in combination with DMT techniques. Technologies like OpenMP [26] and Cilk [29]/Cilk*Plus* [30], offer easy to use interfaces which allow the creation of parallel programs with reduced effort in comparison with the traditional pthreads approach. Mapping existing computational problems in the railway domain to DMT applications is still an open research milestone.

V. Summary and Future Work

This paper presented an introduction to the requirements of safety-critical software in the railway control industry. An industrial example of a widely adopted automation platform (TAS Platform) is given, highlighting architectural characteristics related to transparency and replication. A "formal semantic framework for deterministic parallel programming" [27] is extended in order to show how replica determinism can be achieved by controlling the number of possible schedules in which a parallel application can be executed. It is also discussed how the fork-join model of computation can be achieved in the TAS Platform by making use of the guarantees provided by DMT and StableMT techniques.

Our future work consists of conducting further experimentation with DMT and StableMT approaches in high availability configurations. We will continue to focus on the reconciliation between the properties of parallel programming and the requirements of safety-critical applications in the railway control domain with respect to transparency and replica determinism.

Acknowledgements

This work has been partially supported by the ARTEMIS Joint Undertaking as a part of the EMC2 project under grant agreement no. 621429 and by the Austrian Research Promotion Agency (FFG) project no. 842568.

References

[1] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr, "Basic concepts and taxonomy of dependable and secure computing," in *IEEE Transactions on Dependable and Secure Computing*, vol. 1, no. 1, Jan 2004, pp. 11–33.

[2] H. Kopetz, *Real-Time Systems: Design Principles for Distributed Embedded Applications*, 2nd ed., ser. Real-Time Systems. Springer, 2011.

[3] E. A. Lee, "The problem with threads," *Computer*, vol. 39, no. 5, pp. 33–42, May 2006.

[4] M. Olszewski, J. Ansel, and S. Amarasinghe, "Kendo: Efficient deterministic multithreading in software," in *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XIV. New York, NY, USA: ACM, 2009, pp. 97–108.

[5] J. Yang, H. Cui, J. Wu, Y. Tang, and G. Hu, "Making parallel programs reliable with stable multithreading," *Commun. ACM*, vol. 57, no. 3, pp. 58–69, Mar. 2014.

[6] T. Liu, C. Curtsinger, and E. D. Berger, "Dthreads: Efficient deterministic multithreading," in *Proceedings of the 23rd ACM Symposium on Operating Systems Principles*, ser. SOSP '11. ACM, 2011.

[7] J. Mellado, M. Sierra, A. Romera, and J. Dueñas, "Railway-control product families: The Alcatel TAS Platform experience," in *Software Architectures for Product Families*, ser. Lecture Notes in Computer Science, F. van der Linden, Ed. Springer Berlin Heidelberg, 2000, vol. 1951, pp. 53–62.

[8] A. Gerstinger, H. Kantz, and C. Scherrer, "TAS Control Platform: A platform for safety-critical railway applications," *ERCIM News*, 2008.

[9] *Railway Applications: The specification and demonstration of Reliability, Availability, Maintainability and Safety (RAMS)*, European Committee for Electrotechnical Standardization (CENELEC) Std. EN 50 126-1:1999, 1999.

[10] *Railway Applications: Software for Railway Control and Protection Systems*, European Committee for Electrotechnical Standardization (CENELEC) Std. EN 50 128:2011, 2011.

[11] *Railway Applications: Safety Related Electronic Systems for Signaling*, European Committee for Electrotechnical Standardization (CENELEC) Std., 2002.

[12] *International standard 61508 functional safety: safety related systems*, International Electrotechnical Commission Std. IEC/EN 61 508, 2005.

[13] H. Kantz, S. Resch, and C. Scherrer, "Communication in train control," in *Industrial Communication Technology Handbook, Second Edition*. CRC Press, 2014, pp. 1–15.

[14] J. Pachl, *Railway Operation and Control*, 3rd ed. VTD Rail Publishing, 2015.

[15] J. Palmer, "The need for train detection," in *The 11th IET Professional Development Course on Railway Signalling and Control Systems*, June 2006, pp. 47–53.

[16] H. Pham, "On the optimal design of k-out-of-n:g subsystems," *Reliability, IEEE Transactions on*, vol. 41, no. 4, pp. 572–574, Dec 1992.

[17] ——, "On the estimation of reliability of k-out-of-n systems," *International Journal of Systems Assurance Engineering and Management*, vol. 1, no. 1, pp. 32–35, 2010.

[18] M. Pease, R. Shostak, and L. Lamport, "Reaching agreement in the presence of faults," *J. ACM*, vol. 27, no. 2, pp. 228–234, Apr. 1980.

[19] R. D. Schlichting and F. B. Schneider, "Fail-stop processors: An approach to designing fault-tolerant computing systems," *ACM Trans. Comput. Syst.*, vol. 1, no. 3, pp. 222–238, Aug. 1983.

[20] L. Lamport, R. Shostak, and M. Pease, "The byzantine generals problem," *ACM Trans. Program. Lang. Syst.*, vol. 4, no. 3, pp. 382–401, Jul. 1982.

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

[22] S. Poledna, "Replica determinism in distributed real-time systems: A brief survey," *Real-Time Systems*, vol. 6, no. 3, pp. 289–316, 1994.

[23] S. Adve, "Data races are evil with no exceptions: Technical perspective," in *Communications of the ACM*, vol. 53, no. 11. New York, NY, USA: ACM, Nov. 2010, pp. 84–84.

[24] S. Brilliant, J. Knight, and N. Leveson, "The consistent comparison problem in n-version software," *Software Engineering, IEEE Transactions on*, vol. 15, no. 11, pp. 1481–1485, Nov 1989.

[25] M. McCool, J. Reinders, and A. Robison, *Structured Parallel Programming: Patterns for Efficient Computation*, 1st ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2012.

[26] Openmp. [Online]. Available: http://openmp.org

[27] L. Lu and M. L. Scott, "Toward a formal semantic framework for deterministic parallel programming," in *Proceedings of the Distributed Computing: 25th International Symposium (DISC 2011)*, D. Peleg, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 460–474.

[28] K. Lu, X. Zhou, T. Bergan, and X. Wang, "Efficient deterministic multithreading without global barriers," in *ACM SIGPLAN Notices*, vol. 49, no. 8. ACM, 2014, pp. 287–300.

[29] C. E. Leiserson, "The cilk++ concurrency platform," *J. Supercomput.*, vol. 51, no. 3, pp. 244–257, Mar. 2010.

[30] Cilkplus. [Online]. Available: https://www.cilkplus.org/

[31] A. Aviram and B. Ford, "Deterministic openmp for race-free parallelism." in *HotPar*, 2011.

[32] H. Cui, J. Simsa, Y.-H. Lin, H. Li, B. Blum, X. Xu, J. Yang, G. A. Gibson, and R. E. Bryant, "Parrot: A practical runtime for deterministic, stable, and reliable threads," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, ser. SOSP '13. New York, NY, USA: ACM, 2013, pp. 388–405.

[33] E. D. Berger, T. Yang, T. Liu, and G. Novark, "Grace: Safe multithreaded programming for c/c++," in *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*, ser. OOPSLA '09, vol. 44, no. 10. New York, NY, USA: ACM, 2009, pp. 81–96.

[34] T. Merrifield, J. Devietti, and J. Eriksson, "High-performance determinism with total store order consistency," in *Proceedings of the Tenth European Conference on Computer Systems*, ser. EuroSys '15. New York, NY, USA: ACM, 2015, pp. 31:1–31:13.

[35] T. Merrifield and J. Eriksson, "Conversion: Multi-version concurrency control for main memory segments," in *Proceedings of the 8th ACM European Conference on Computer Systems*, ser. EuroSys '13. New York, NY, USA: ACM, 2013, pp. 127–139.

[36] C. Directive, "96/48/ec of 23 july 1996 on the interoperability of the trans-european high-speed rail system," *Official Journal L*, vol. 235, no. 17, p. 09, 1996.

[37] Wikipedia, "European train control system — wikipedia, the free encyclopedia," 2016, [Online; accessed 11-February-2016]. [Online]. Available: https://en.wikipedia.org/w/index.php?title=European_Train_Control_System&oldid=704210769

[38] K. Lakshmanan, S. Kato, and R. Rajkumar, "Scheduling parallel real-time tasks on multi-core processors," in *Real-Time Systems Symposium (RTSS), 2010 IEEE 31st*, Nov 2010, pp. 259–268.

[39] C. Maia, L. Nogueira, and L. Pinho, "Scheduling parallel real-time tasks using a fixed-priority work-stealing algorithm on multiprocessors," in *Industrial Embedded Systems (SIES), 2013 8th IEEE International Symposium on*, June 2013, pp. 89–92.