A Methodology for the Design and Deployment of Reliable Systems on Heterogeneous Platforms

Hugo A. Andrade*†, Arkadeb Ghosal*, Kaushik Ravindran* and Brian L. Evans†
*National Instruments Corporation, Berkeley, CA 94704, USA
†Dept. of Electrical and Computer Engineering, The University of Texas at Austin, Austin, TX 78712, USA
E-mail: {hugo.andrade, arkadeb.ghosal, kaushik.ravindran}@ni.com, bevans@ece.utexas.edu

***Abstract*—Heterogeneous multi-target platforms composed of processors, FPGAs, and specialized I/O are popular targets for embedded applications. Model based design approaches are increasingly used to deploy high performance concurrent applications on these platforms. In addition to programmability and performance, embedded systems need to ensure reliability and availability in safety critical environments. However, prior design approaches do not sufficiently characterize these non-functional requirements in the application or in the mapping on the multi-target platform. In this work, we present a design methodology and associated run-time environment for programmable heterogeneous multi-target platforms that enable design of reliable systems by: (a) elevating reliability concerns to the system modeling level, so that a domain expert can capture reliability requirements within a formal model of computation, (b) modeling platform elements that can be automatically composed into systems to provide a reliable architecture for deployment, and (c) segmenting (in space and time) the run-time environment such that the system captures independent end-user provided reliability criteria. We illustrate the modeling, analysis, and implementation capabilities of our methodology to design fault tolerant control applications. Using the National Instruments PXIe platform and FlexRIO components, we demonstrate a run-time environment that provides desired levels of reliability.**

I. Introduction

Heterogeneous programmable multi-target platforms are an established trend in mainstream and embedded computing [1], [2]. Such platforms are composed of multiple processors, FPGAs, GPUs, application specific processing units, distributed memories, and specialized interconnection networks and I/O. The continuous increase in performance requirements of embedded applications has fueled the advent of such high-performance platforms. At the same time, the need to adapt products to rapid market changes has made programmability an important criterion for the success of these devices. These platforms are dominant in a variety of markets including digital signal processing, gaming, graphics, and networking. The recent surge of applications in medicine, finance, and security will only encourage the proliferation of these platforms.

In addition to the need for programmability and performance, embedded systems need to ensure higher reliability and availability in safety critical environments [3]. These systems are often large and complex and downtime is very expensive [4]. This makes reliability an increasingly important design consideration. The new technologies, owing to the harsh nature of the deployment conditions, are more susceptible to hard errors due to external effects like radiation, catastrophic failures, malicious attacks, and deterioration over time. The complexity of the design solution makes them increasingly prone to manufacturing (lower yield) errors, design errors, and run-time (soft) errors. The typical platform is therefore a collection of unreliable computing devices, communicating over an unreliable network, and surrounded by I/O that may be unreliable as well. The application components deployed on this platform may also be subject to design and run time errors, or even contain maliciously created programs.

The larger number and relative independence of multiple processing components motivates a design approach that exploits redundancy to improve reliability and availability. The system can be viewed as a collection of possibly redundant computation, communication, and I/O components, which can be replaced in the field while the system is in operation. The redundancy of these field replaceable units (FRUs) are exploited to maintain the mean time between failure (MTBF) within reasonable boundaries and create a dynamic fault-tolerant system. FRUs are typically low cost parts, composed of few components, as opposed to an entire chassis, which helps minimize the cost of repair and replacement.

Building a reliable, efficient, and cost-effective system of FRUs brings forth new challenges. The application specification must be extended to include non-functional requirements related to reliability and availability. The platform model should capture the reliability of individual components and define how the reliability of the combined system is evaluated. Further, in most embedded applications, the nature and affinity of I/O is an important consideration. FRUs usually have direct interaction with the I/O of the system. The processing components of these FRUs are complemented by reconfigurable devices that serve as interfaces to the I/O and guarantee control and timing requirements. This introduces additional constraints that regulate how these FRUs can be reconfigured and computation can be re-distributed to exploit redundancy. The challenge then is to develop models, tools, and run-time environments for the design and deployment of reliable fault-tolerant systems on heterogeneous multi-target platforms.

In this paper, we advance a design methodology that elevates reliability concerns to the system level and propose a framework that enables reliable system level design by application domain experts. The salient features of our methodology are: (a) *specification of non-functional requirements* in terms of in-

978-1-4673-2921-7/12/$31.00

tuitive and well-defined models of computation and application design patterns; (b) *availability of dynamic virtual platform infrastructure* based on coarser granularity reliable elements on top of programmable components; (c) and, *hardware-software co-design environment* that uses dynamic mapping to adapt to different operational scenarios.

Our methodology is inspired by the Y-chart approach for system design and design space exploration (DSE) [2], [5], [6]. The Y-chart approach advocates a deliberate separation of concerns related to application specification, platform model, and mapping between them. We advocate that reliability and availability concerns be emphasized at all steps in the Y-chart methodology. The non-functional requirements must be formally encoded in the application specification. Similarly, the redundancy and replaceability of the computation and I/O components must be explicit in the platform model. The mapping step binds the application to the architectural resources while enforcing non-functional requirements. This methodology emphasizes the productivity of domain experts who are knowledgeable in a technical field but not necessarily adept in deploying embedded systems. The goal is to help the application domain expert design a reliable fault-tolerant distributed embedded system, while being able to focus on important design problems. The methodology ensures that the issues of reliability are not ignored, while obviating the expert from having to delve into the details of reality of deploying a distributed computing environment.

In this work, we demonstrate our methodology for the design of reliable systems on multi-target platforms by applying it to the National Instruments (NI) LabVIEW Graphical System Design framework [7]. LabVIEW targets a platform consisting of (reconfigurable) measurement and control I/O modules, which can be aggregated into cyber-physical systems by combining the I/O modules with industrial controllers or external computers. The framework includes specialized add-ons like LabVIEW RT [8] (for real-time systems), LabVIEW FPGA [9] (for reconfigurable platforms with FPGAs), and StateCharts development module [10] (for state transition systems). The framework organization and flow (Fig. 1) closely resembles the Y-chart methodology discussed in [11] and [2]. Traditionally, LabVIEW has focused on functional modeling. In this work, we extend LabVIEW to include non-functional requirements related to reliability and availability and present case studies to validate the design methodology.

Section II compares related methodologies for the design of reliable systems. The proposed framework based on Y-chart methodology is reviewed in Section III. Section IV discusses patterns to capture fault tolerance requirements that serve as examples for the case study. Section V demonstrates a prototype run-time infrastructure and presents the case study. Section VI concludes with future directions.

II. Related Work

Synthesis of reliable systems from higher level specifications has been well studied in the literature [12] [13] [14]. These researchers have extended conventional models of

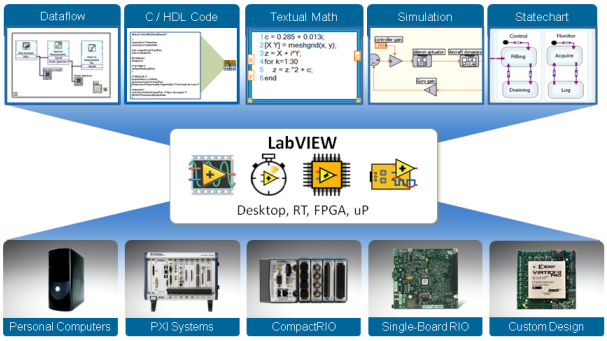

Fig. 1. Platform-based Design and MoCs

computation (e.g. dataflow or timed interaction of tasks) to specify reliability requirements. As discussed in Section I, the LabVIEW graphical system design framework (Fig. 1) supports multiple models of computation (dataflow, math, textual etc.) for functional aspects that can be synthesized over multiple hardware platform elements (behavioral polymorphism [15]). Recent efforts have extended the framework with capabilities to capture non-functional requirements, and allow subsequent synthesis on heterogeneous platforms. [16] reports extensions to allow non-functional specifications like throughput and latency for communications applications deployed on reconfigurable targets. In this work, we enhance the LabVIEW framework to support the capture of non-functional requirements like reliability, and enable system synthesis from these requirements. In particular, we focus on pattern-based specification of fault-tolerant systems [3] and on templates for patterns in non-functional requirements [17].

Our methodology supports synthesis for heterogeneous computation, communication, and I/O elements. [18] [19] present prototype run-time environments that target processors and reconfigurable elements with standardization at the operating system (OS) level. We provide support at three levels: the ability to capture system level requirement with reliability constraints, the ability to integrate third party IP (possibly using [20] [21]), and the ability to support multiple commercial hardware platforms.

A key aspect of the run-time infrastructure for fault tolerance is the ability to adapt to diverse requirements. Prior works referred here use high level specifications at the system level, but there is a disconnect between the model and the run-time environment. [22] presents an interesting approach where a run-time optimizer has been used with early design information for better adaptive behavior. Our goal is to implement a secondary run-time optimizing mapping engine that complements the initial system level mapping with updates to the platform model in order to provide adaptive behavior. In summary, the methodology advanced in this paper attempts to integrate the respective strengths of the previously discussed tools into a unified framework for implementation of reliable systems onto heterogeneous platforms, while maintaining an intuitive graphical design environment for domain experts who may not be experts in hardware design.

III. Flow

As discussed in the introduction, we follow the Y-chart design/synthesis methodology (Fig. 2) which begins with separate specifications of application and architecture. An explicit mapping step binds the application to the architectural resources. The result of the evaluation (of the mapping) forms the basis for further mapping iterations: the designer has the choice to modify the application and the workload, the selection of architecture building blocks, or the mapping strategy itself. Thus, the Y-chart enables the designer to methodically explore the design space of system implementations [6].

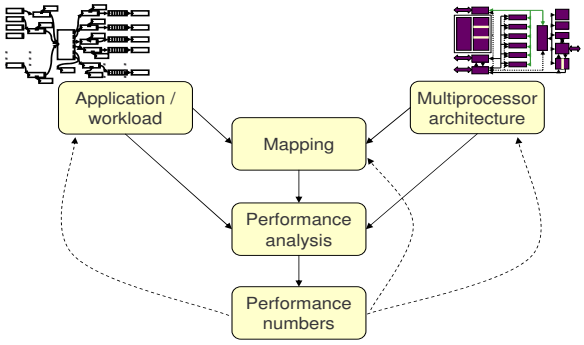

Fig. 2. The Y-Chart approach for deploying concurrent applications.

The goal of the mapping step in the Y-chart framework is to maximize application performance (or some other design objective) by an effective distribution of computation and communication on to the resources of the target architecture. Thus, to map the task level concurrency in an application to a multi-target platform entails solving a complex multi-objective combinatorial optimization problem subject to several implementation and resource constraints. A manual designer-driven approach, based on commonly accepted "good practice" decisions, may be satisfactory today; but the relative quality of solutions will only deteriorate in the future as the complexity of applications and architectures continues to increase. Therefore, it is less and less likely that a good design is a simple and intuitive solution to a design challenge [6]. This motivates the need for an automated and disciplined methodology to guide a designer in evaluating large design spaces and creating successful system implementations.

The objective of this work is to create an automatic tool flow that handles specification of application (with functional and non-functional requirements) and hardware platform (with reliability metrics), computation/evaluation of possible mappings for different optimization objectives, and synthesis/implementation of the application on the hardware platform based on the selected mapping. In this paper we focus on two aspects of the design flow. Section IV discusses how reliability/availability requirements can be mapped from application to platform using pattern information assuming MTTF/MTBF is available for the platform. Section V presents a prototype run-time implementation of the patterns discussed in Section IV. Two other aspects, a specification language (to formally express reliability patterns) and an automated mapping algorithm (to detect the best possible assignment of tasks to processors), have been partially documented in [23] and [24] and are the subject of ongoing research.

IV. Pattern Specification

A fault in a system gives rise to an error. The basic idea of fault tolerance is to detect the error and to take necessary action to process the error through recovery and/or mitigation. Once error processing completes, the system returns to normal operation. If needed, the system can perform fault treatment during normal operation, to prevent the error from recurring. Depending upon scenarios and needs of the system, error detection, recovery, and mitigation can be classified into patterns. Patterns for fault tolerance have been well studied; refer to [3] for a comprehensive list of different patterns for error detection, recovery, and mitigation. In this section we discuss one pattern each for error detection, recovery, and mitigation using reconfigurability to implement the patterns. Reconfigurability is used to reduce the overall cost of the system, and to maximize the use of resources. We use the following patterns:

- *Watchdogs* (for Error Detection) uses special entities to watch over the system (or specific components) to make sure that the system is operating well.
- *Failover* (for Recovery) recovers by switching to a redundant unit.
- *Shed Load* (for Mitigation) discards some requests for service to offer better service to other requests.

We investigate the possibility of using reconfigurability to avoid a *standby* redundant unit which reduces the overall cost. In particular, we show that both computation and communication can be reconfigured on-demand based on the requirements of availability and reliability. The following sections define reliability and availability metrics and discuss the patterns.

A. Reliability and Availability Metrics

Reliability (R) of a component is the probability that the component will perform the intended function during a given period of time under specified operating conditions. Reliability is computed as $R = e^{-\lambda * t}$, where λ is the inverse of $MTTF$ (*Mean Time To Failure*), t is the period of time for which reliability is being calculated, and $MTTF$ is the expected time to failure for the component. For n components, reliability of a system with the components in series (resp. parallel) is $\prod_{i=1}^{n} R_i$ (resp. $1 - \prod_{i=1}^{n}(1 - R_i)$), where R_i is the reliability of the i-th component.

Availability (A) of a component is the probability that a component will perform the intended function over a period of time when operated and maintained in a prescribed manner. Given $MTTF$ and $MTTR$ (*Mean Time To Repair*), availability $A = \frac{MTTF}{MTTF+MTTR}$ where $MTTR$ is the expected time to repair/restore the component. When n replicas of a component are in series (resp. parallel), availability of the system is A^n (resp. $1 - (1 - A)^n$).

B. Increasing Availability without Explicit Backup Redundancy

In this pattern, reconfigurability is used to meet availability requirements. Two tasks $T1$ and $T2$ are running on two reconfigurable processors $F1$ and $F2$ respectively. $T1$ has higher priority than $T2$ in terms of availability, and requirement for the design states that if at least one of the processors is operational, then $T1$ must be executing. The processing requirements of the tasks are such that both cannot be placed simultaneously on the same processor. Fig. 3 shows the states of the system in terms of mapping between tasks and processors. In states 2 and 3, only one processor is operational, and task $T1$ executes on the operational processor. When the other processor is back to operational (in states 2 or 3), $T2$ is reconfigured on that processor, and the system has both tasks running (state 4 or 1). As long as either $F1$ or $F2$ is operational the downtime for $T1$ is only the reconfiguration time if the task needs to be moved. The pattern can be extended for more than two tasks (with different levels of priorities) executing on multiple reconfigurable processing units. If availabilities of the processors are 0.998 each, then availabilities of the tasks are 0.998 each as long as they are bound to one processor only. If task $T1$ can be moved back and forth between the two processors, then the availability of task $T1$ improves to 0.999996.

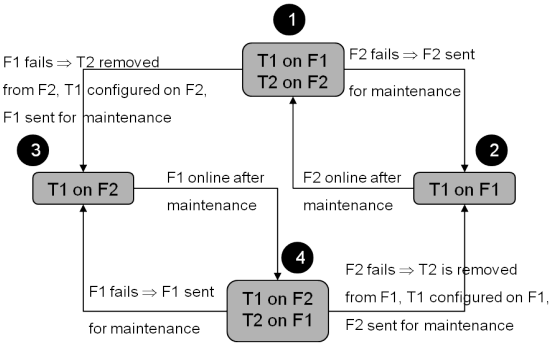

Fig. 3. Pattern 1

The time to repair can be refined as $MTTR = MTTI + MTTX + MTTD$, where $MTTI$ (*Mean Time To Identify*) is the time to detect whether the failure is transient or semi-permanent/permanent, $MTTX$ (*Mean Time To Fix*) is the time to fix and/or replace a component that has a failure, and $MTTD$ (*Mean Time To Deploy*) is the time to deploy/reconfigure a task on a processor. Note that $MTTI$ and $MTTD$ may be orders of magnitude smaller than $MTTX$, thus making the average value of $MTTR$ smaller relative to the time to repair/fix/replace a failed component.

C. On-demand Reliability based on Modal Requirements

Similar to the last pattern, there are two tasks ($T1$, $T2$) and two reconfigurable processors ($F1$, $F2$). There are two operation modes $m1$ and $m2$ (see Fig. 4). In mode $m1$, tasks $T1$ and $T2$ execute on $F1$ and $F2$, respectively. In mode $m2$, task $T2$ is removed and task $T1$ is replicated on $F1$ and $F2$ such that $T1$ runs in DMR (Dual Redundancy Mode). This implies task $T1$ runs with higher reliability in mode $m2$ than in mode $m1$. The mode switch is triggered by an external monitor based on user command or change in environment. Mode $m1$ can switch to mode $m2$ only if both processors are operational. If either of the processors fails in mode $m2$, the remaining operational processor will continue to run task $T1$.

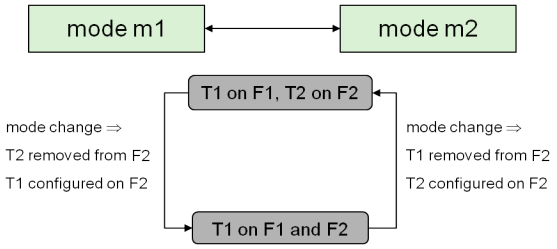

Fig. 4. Pattern 2

When mode $m1$ switches to mode $m2$, both computation and communication need to be reconfigured. Computation needs to be reconfigured as task $T1$ need to be replicated. Communication needs to be reconfigured as results from the processors should be sent for voting, which can be implemented on the embedded controller, on the running processors, or on a third processor. Care should be taken such that voting is not the bottleneck, which may reduce the overall reliability. If reliabilities of the processors are 99% each, then in mode $m1$ both tasks have reliability of 99%, while in mode $m2$, reliability of task $t1$ is 99.99%. The processing units have been assumed to be a *black box* system. However processing units may consist of core (FPGA), memory, I/O conditioning units, and communications interfaces. If reliabilities for individual components are available, then they should be factored in the calculation. The pattern can be extended for more than two tasks running on more than two processing units. E.g., in a platform with three processing units, a task may be configured to run solo or in DMR or in TMR (Triple Modular Redundancy) as needed for scenarios by replacing lower priority tasks.

V. Case Study

As a step towards building a tool that supports the flow described in Section III, we prototyped parts of the run-time infrastructure. Though the flow presented here is manual, the overall methodology is inspired by the Y-chart approach. Fig. 5 shows the application, platform, and system view. The application consists of task definitions with I/O drivers for each task. In the case study, there are two tasks $T1$ and $T2$ with two drivers $d1$ and $d2$ respectively. The platform consists of one embedded controller (implementing the watchdog), two modular computing nodes with FPGAs, and a backplane supporting communication between the controller and the computing nodes. The computing nodes connect to I/O via drivers. The system supports links from both I/O to the two computing nodes. The links may not be working all the time;

a link would be activated as required to implement a scenario for a specific pattern.

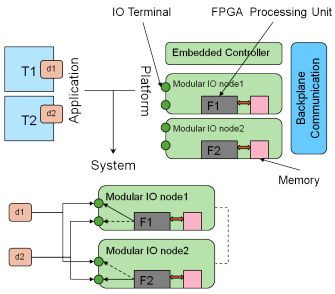

Fig. 5. Design Flow

For the experiments, we have used some of the platform elements currently available with the National Instruments PXI Express (PXIe) platform [25] (as shown in Fig. 6):

- **PXIe-1075** (marked as *PXI Express Chassis*) is a 3U 18-slot chassis with up to 1 GB/s slot-to-slot dedicated bandwidth and 4 GB/s system bandwidth, via a PCI Express based interconnect.
- **PXIe-8133** (marked as *RT Host Controller*) is a real-time (RT) quad-core Intel Core i7-820 based controller, with a high-bandwidth PXI Express embedded interface which supports up to 8 GB/s of system and 2 GB/s of slot-to-slot bandwidth.
- **PXIe-7965R** (marked as *RIO1* and RIO2) is a FlexRIO FPGA module with a Virtex-5 SX FPGA optimized for digital signal processing with onboard RAM, and DMA channels for high-speed data streaming at more than 800 MB/s. General purpose I/O lines are accessible through the front panel of the FPGA module and serve as interconnections to optional Front Adapter Modules (the NI-5781 baseband transceiver in this case).

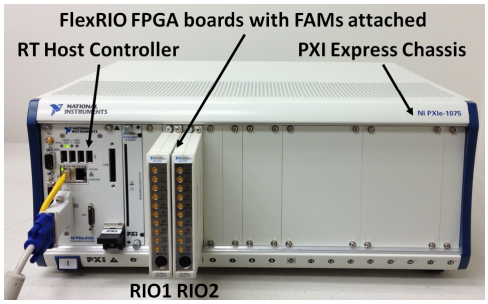

Fig. 6. PXIe-based Platform

The PXIe platform is used not only in test and measurement applications, but also in embedded high performance applications. A similar platform configuration as the above has been used at CERN [4] to provide reliable measurements and control for the collimators at LHC.

Some of the platform elements like the chassis already have specific features to increase availability at the system level. E.g. the PXIe-1066DC chassis incorporates hot-swappable, redundant DC power supplies and hot-swappable, front-accessible, redundant cooling fans to maximize system availability. Reliability issues of PCI Express have been discussed in [26]. As mentioned in the previous section for now we assume the interconnect is 100% reliable.

With the proposed methodology we are exploring a finer granularity for the FRUs compared to the current chassis level, and focussing on the reliability closer to the I/O ports of the system, so that we would need to only replace individual boards. We take advantage of some of the independent slot-to-slot communication existent in PCIe, and the redundancy and reconfigurability of the I/O platform elements to implement parts of the reliable run-time support.

A. Run-time Infrastructure Prototype to Increase Availability without Explicit Replication

We first prototyped the infrastructure required to implement a system using the first pattern described in section IV, where we give priority to the execution of task T1 over task T2 on a platform that has two processing engines (in this case the FPGAs labeled RIO1 and RIO2). An implementation of tasks T1 and T2 in the LabVIEW FPGA (G) graphical programming language are shown in Fig. 7. G provides an actor-based structural homogeneous dataflow model of computation [27]. In this case the tasks are providing two levels of service of a simple linear computation of the data coming from the on-board ADCs, which is then sent to the real-time controller via DMA channels.

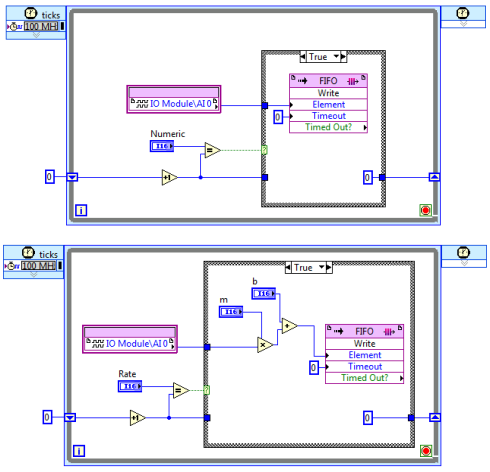

Fig. 7. LV FPGA Task Block Diagrams (T1 top, T2 bottom) for Pattern 1

Fig. 8 shows the host computer implementation. As shown in the yellow comments, we first execute one iteration of the state machine. The system state is maintained by the LabVIEW StateCharts and closely resembles the high level description in Fig. 3. The input to the state machine includes failure and repair flags for each of the FPGA boards. The host program refers to the output of the state machine, and performs the necessary reset, replacement or reconfiguration. Although not explicitly modeled for this case study, we can include a fault identification step to differentiate between

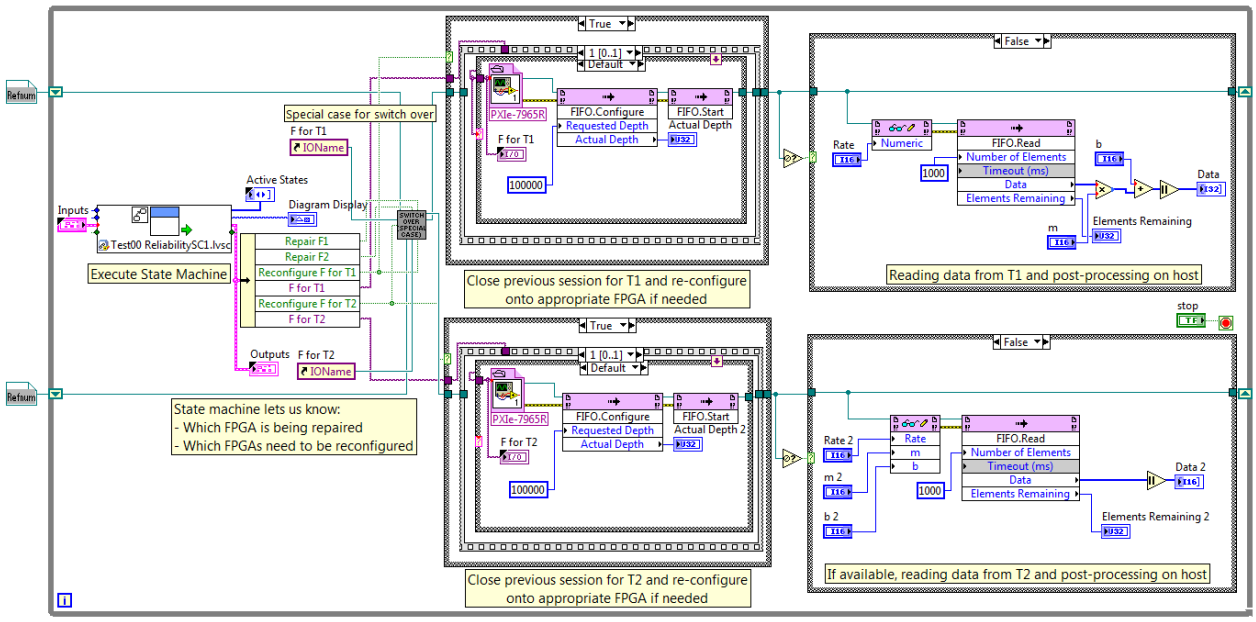

Fig. 8. LV Host Block Diagram for Pattern 1

transient failures (which would require a reconfiguration) and permanent failures (which would only require component replacement). An operator would be alerted to the fact that a repair is necessary. To control the life cycle of the task on a FPGA I/O board we use the LabVIEW FPGA Host API. In addition to basic dynamic downloading capabilities, it provides reset, abort, and control/status register access, as well as high-performance streaming interfaces. The framework does not provide partial configuration capabilities yet, so a full reconfiguration is required to download a new task onto the FPGAs. For our implementation, the absence of partial reconfiguration capability does not impose significant restrictions, but we would be able to take advantage of such feature, when available, as partial reconfiguration can reduce MTTD.

Fig. 9 shows the front panel or console used to control the experiment. On the left side we see the data captured by T1 and T2, as well as the FPGA on which they were last running, depending on which one was operational. For the experiment, we can also override the error detection on each FPGA, and we are alerted when an FPGA needs to be serviced, and we can let the system know that an I/O board has been repaired. The right side of the figure depicts the state transition of the system depending on failures of the modules RIO1 or RIO2.

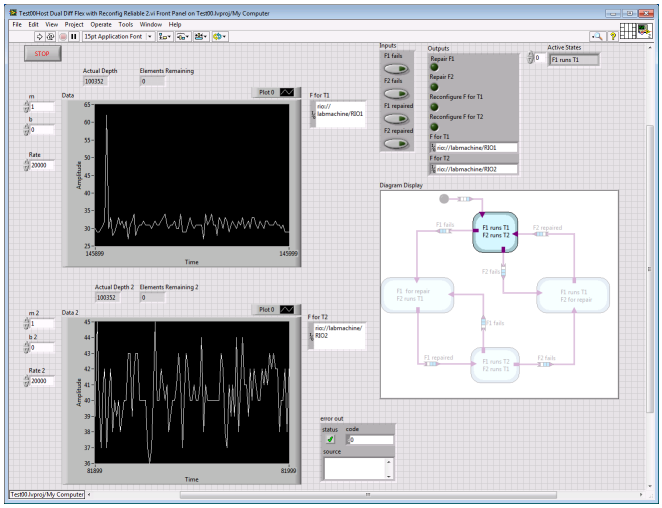

Fig. 9. LV Host Front Panel for Pattern 1

B. Run-time Infrastructure Prototype for On-demand Reliability based on Modal Requirements

We also prototyped the run-time environment to implement a DMR scheme between two RIO FPGA boards, which check each other for possible errors. This resembles mode $m2$ of the patterns shown in Section IV. We have not implemented both the modes and mode switch as we have already shown the possibility of switching while discussing the first pattern. Hence this subsection only focuses on implementing the DMR scheme. If an error is identified, then the host will be alerted to read the functional FPGA. The scheme can be extended to TMR if three FPGA boards are available. Each of the FPGAs implements the code shown in Fig. 10. In the code, references for data to and from the other/redundant unit are labeled with 'RU'. With dual modular redundancy we can identify a problem, but cannot provide fail safe implementation. To avoid the limitation, we have implemented a set of heuristics to better predict which stream the host should read from. For each RIO board we send a copy of the data stream (or arbitrarily sub-sampled version to avoid overhead) to the redundant unit via a peer-to-peer connection. There it is synchronized with its own data stream and compared in the actor labeled 'T-DMR

Controller'. This would indicate possible errors at the source of data, and based on local and remote (which are passed through another peer-to-peer link from the RU) diagnostic heuristics we identify the board that is in better condition to generate data for the host to continue computation.

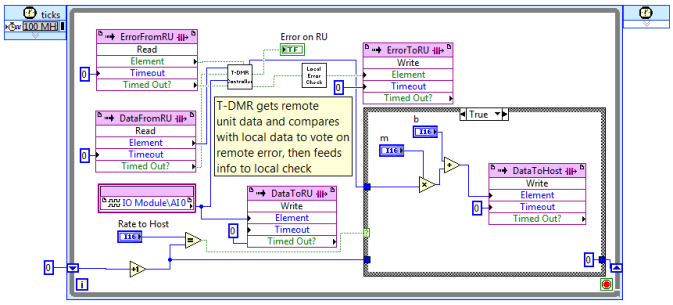

Fig. 10. LV FPGA DMR Block Diagram for Pattern 2

The RIO board to Host communication is done via a DMA stream from each board. As shown in Fig. 11, at each iteration of the host software, we check which stream the host should be reading from. Given that for synchronization purposes we have time stamps for the data, the information can be passed to the host, which can then (during switch-over between I/O sources) reorganize the data or ignore out of order data with very little overhead. We can also proceed to reconfigure or repair/replace the faulty board while operating from the redundant unit.

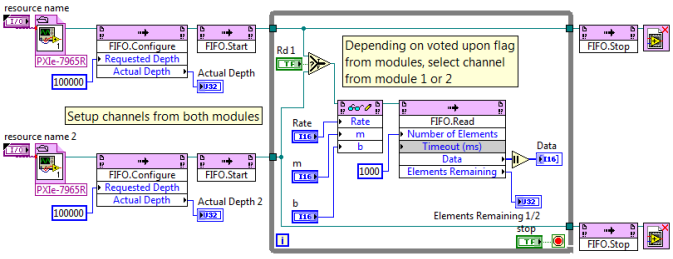

Fig. 11. LV Host Block Diagram for Pattern 2

VI. Conclusions

In this work, we focused on two aspects of design and deployment of reliable systems on heterogeneous platforms: (1) we showed examples of how reliability/availability requirements can be mapped from application to platform using pattern information for given MTTF/MTBF information for a platform, and (2) we used the NI PXIe platform and FlexRIO components to demonstrate the viability of a run-time environment that provides desired levels of reliability for two requirement patterns.

In the future, we intend to focus on (a) formalizing pattern specification language, (b) modeling platform elements based on reliability, and (c) defining techniques to map functional requirements on platform based on reliability requirements. In the end, we would like to combine the above into a consistent automatic flow for synthesis of reliable systems on heterogeneous multi-target platforms.

References

[1] A. L. Sangiovanni-Vincentelli, "Quo Vadis SLD: Reasoning about Trends and Challenges of System-Level Design," *Proceedings of the IEEE*, vol. 95, no. 3, March 2007.

[2] K. Keutzer, S. Malik, A. R. Newton, J. Rabaey, and A. Sangiovanni-Vincentelli, "System Level Design: Orthogonalization of Concerns and Platform-Based Design," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 19, no. 12, Dec 2000.

[3] R. S. Hanmer, *Patterns for Fault Tolerance Software*. Wiley Series in Software Design Patterns, 1978.

[4] A. Masi, A. Brielmann, R. Losito, and M. Martino, "LVDT Conditioning on the LHC Collimators," *IEEE Trans. on Nucl. Sci.*, vol. 55, 2008.

[5] B. Kienhuis, E. F. Deprettere, P. v. d. Wolf, and K. A. Vissers, "A Methodology to Design Programmable Embedded Systems - The Y-Chart Approach," in *Embedded Processor Design Challenges: Systems, Architectures, Modeling, and Simulation - SAMOS*, 2002.

[6] M. Gries, "Methods for Evaluating and Covering the Design Space during Early Design Development," *Integr. VLSI J.*, vol. 38, no. 2, Dec 2004.

[7] "LabVIEW System Design Software," www.ni.com/labview/.

[8] "NI LabVIEW Real-Time Module," www.ni.com/labview/realtime/.

[9] "NI LabVIEW FPGA Module," www.ni.com/labview/fpga/.

[10] "NI LabVIEW Statechart Module," www.ni.com/labview/statechart/.

[11] E. Lee and S. Neuendorffer, "Concurrent Models of Computation for Embedded Software," *Tech. Memorandum UCB/ERL M04/26, EECS, Univ. of California, Berkeley*, 2004.

[12] C. Pinello, L. P. Carloni, and A. L. Sangiovanni-vincentelli, "Fault-Tolerant Deployment of Embedded Software for Cost-Sensitive Real-Time Feedback-Control Applications," in *Proc. of Design Automation and Test in Europe*, 2004.

[13] C. Buckl, D. Sojer, and A. Knoll, "FTOS: Model-driven Development of Fault-tolerant Automation Systems," in *Proc. of the IEEE International Conference on Emerging Techonologies and Factory Automation*, 2010.

[14] K. Chatterjee, A. Ghosal, T. A. Henzinger, D. Iercan, C. M. Kirsch, C. Pinello, and A. Sangiovanni-Vincentelli, "Logical Reliability of Interacting Real-time Tasks," in *Proc. of Design Automation and Test in Europe*, 2008.

[15] E. A. Lee and Y. Xiong, "A Behavioral Type System and its Application in Ptolemy II," *Formal Aspects of Computing*, vol. 16, no. 3, 2004.

[16] H. Andrade, J. Correll, A. Ekbal, A. Ghosal, D. Kim, J. Kornerup, R. Limaye, A. Prasad, K. Ravindran, T. N. Tran, M. Trimborn, G. Wang, I. Wong, and G. Yang, "From Streaming Models to FPGA Implementations," in *Proc. of the Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA)*, 2012.

[17] A. Armoush, F. Salewski, and S. Kowalewski, "Design Pattern Representation for Safety-Critical Embedded Systems," *Journal Software Engineering and Applications*, vol. 2, no. 1, 2009.

[18] H. K. So, A.Tkachenko, and R. Brodersen, "A Unified Hardware/Software Runtime Environment for FPGA-based Reconfigurable Computers using BORPH," *Proc. of the 4th international Conference on Hardware/Software Codesign and System Synthesis*, 2006.

[19] X. Iturbe, K. Benkrid, A. T. Erdogan, T. Arslan, M. Azkarate, I. Martinez, and A. Perez, "R3TOS: A Reliable Reconfigurable Real-Time Operating System," *Proc. of NASA/ESA Conference on Adaptive Hardware and Systems*, 2010.

[20] "IP-XACT," www.accellera.org/activities/committees/ip-xact/.

[21] "OpenCPI - Open Component Portability Infrastructure," opencpi.org/.

[22] P. Benoit, "Distributed Approaches for Self-Adaptive Embedded Systems," in *Proc. of the Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA)*, 2012.

[23] H. Andrade, A. Ghosal, and K. Ravindran, "Pattern-based Specification of Non-Functional Requirements and Synthesis onto Dynamically Reconfigurable Platforms," in *IEEE International Reliability Innovations Conference (IRIC)*, 2012.

[24] H. A. Andrade and K. Ravindran, "Automatic System-Level Synthesis for Heterogeneous Platforms," in *Proc. of the IEEE Real-Time Conference*, 2012.

[25] "NI PXI Platform," www.ni.com/pxi/.

[26] K. Kong, "PCIe as a Multiprocessor System Interconnect," *http://www.pcisig.com*, 2012.

[27] H. A. Andrade and S. Kovner, "Software Synthesis from Dataflow Models for G and LabVIEW," in *Proc. of the IEEE Asilomar Conference on Signals, Systems, and Computers*, 1998.