Property Preservation and Composition with Guarantees: from ASSERT to CHESS

(Invited Paper)

Tullio Vardanega

Dept. of Pure and Applied Mathematics, University of Padua, Italy
Email: tullio.vardanega@math.unipd.it

Abstract

While the demand for high-integrity applications continues to rise, industrial developers seek cost effective development strategies that are capable of delivering the required guarantees. The very nature of high-integrity software systems make a-posteriori verification totally inapt to meet the time, cost and quality constraints that impend on developers. What is wanted instead is a development method that facilitates early verification and that devolves to proven automation as many of the error-prone development tasks as practically possible. Model-driven engineering (MDE) is an especially fit option to explore in that respect.

In a recent European project very interesting results were obtained in the development and industrial evaluation of an MDE process centered on the joint principles of correctness by construction and property preservation. The proceedings of that project were so encouraging in fact that a continuation of it was instigated with a challenging broader scope. This paper provides an account of the approach taken in the original project with regard to property preservation and outlines the intent of its continuation.

1. Introduction

For an increasing number of application domains the guarantee of high-assurance operation is a paramount obligation vested on the supplier. Software systems that qualify for use in those domains are generally referred to as high-assurance or equivalently as high-integrity systems. (In the following we shall use the latter nomenclature in preference.)

To characterize the demands that are placed on high-integrity systems, one could use a parallel to the notion of service-level agreement (SLA) that stands central to the delivery and operation of quality-controlled IT infrastructures [1]. In a way, every individual software component of a high-integrity system as well as their full integration are subject to specific SLA that prescribes the functional quality of the services to be delivered, the non-functional properties to be exhibited and the mitigation behavior to take upon failures.

The very notion of high-integrity system is a most extreme antithesis to the "fail first patch later" paradigm that is typical of general-purpose off-the-shelf computing systems. It is for this reason that, as it was put in [2], one of the major challenges in the engineering of high-integrity software is to develop well-founded methods for system construction, verification and validation.

Two important consequences follow from the cited observation: (i) a-posteriori verification is largely inadequate, in time, cost and effectiveness, to build the required level of confidence in the correctness and quality of the services delivered by the system software; (ii) the "high-integrity" attribute of a software system is obtained by a combination of process and product characteristics.

The challenge that ensues from both respects still stands to date, though techniques have emerged that help tackle it.

One of the most tempting and perhaps most obvious solutions is to take stock of best practices in the two dimensions of process and product and explore what industrially applicable solutions could be developed on their basis.

In our contention, a twofold goal should drive the quest for solutions: (i) to establish correctness as early as possible; and (ii) to actively preserve it throughout the whole development process down to deployment and execution.

In the industrial strive for the cost effectiveness of software development, the greater levels of automation attainable by model-driven engineering (MDE) [3], [4] are particularly attractive. If MDE technologies and methodologies could be devised to serve the needs of high-integrity systems, then correctness should be proven at the level of the user model while its preservation should be warranted by the fabric of automation.

The good news is that both objectives can be reached. Some evidence begins to emerge to that effect, as for example from the ASSERT project (*Automated proof-based System and Software Engineering for Real-Time systems* [5]). ASSERT was, from its onset, aimed to define an MDE methodology and the associated infrastructure suited for the production of high-integrity software systems for use in space applications. In its 40-month lifespan it achieved important results in that direction. This paper reports on some of the premises and achievements of the ASSERT project, especially along the axis of preserving correctness throughout the whole extent of automation, deployment and execution. In fact, the ASSERT results were so encouraging that a core subset of its project team instigated the launch of a successor project (nicknamed CHESS and funded in the ARTEMIS JU framework [6]) that shall pursue two

additional goals in a broader cross-section of high-integrity industry spanning railway, space, telecommunications and, possibly, automotive: the preservation of dependability properties in an ASSERT-like approach; and the guarantee of compositionality in the user modeling space so that component reuse may be more intensively and proficiently fostered in high-integrity applications.

The remainder of this paper is organized as follows: Section 2 recalls the goals that were set on ASSERT; Section 3 outlines the key elements of the ASSERT approach; Section 4 describes the infrastructural elements which most contribute to the property preservation qualities of the ASSERT development method; Section 5 discusses the coverage of system properties achieved by combination of proven transformations and property-preserving execution; Section 6 illustrates the main steps of the transformation process; Section 7 finally draws some conclusions and anticipates the lines of development that will be pursued in the CHESS project in continuation of ASSERT.

2. The ASSERT Goals

Table 1 summarizes the goals that the ASSERT project was to meet. Let us briefly review them in isolation.

High-level goal	Breaks down into	Applies to
Separation of Concerns		Process
Property Preservation		Process
		System
High Integrity		Process
	Timing predictability	System
	Isolation	System
Distribution Transparency		System

Table 1. ASSERT goals.

Separation of Concerns. One of the founding principles of the ASSERT approach was the strive for separation of concerns. The notion of separation of concerns is fairly old as it dates back to [7] in 1974: unfortunately, in the meanwhile it has been misunderstood, abused and frequently dispensed with. The advent of MDE however eases the definition of concern-specific views and helps prove the benefit that their separation may bring.

In the project vision, the main axis of separation is between functional and non-functional: the former is where the product-specific algorithmic value added is, while the latter is best addressed by the guided deployment of proven recurrent solutions suitably represented in terms of predefined, configurable and composable components, patterns and archetypes.

In relation to the general context of component-based development [8] and the wealth of methodologies, techniques and formalisms that have been proposed to formalize component-based design (cf. e.g.: [9]–[13]) the ASSERT approach as outlined in [14] sides with the constructive approaches on account of its restrictive and specialized choices, with the bonus of proving stronger results that are not true in the general case.

Of all possible dimensions of non-functional concerns, ASSERT focused on predictability and isolation and with the ambition to use a model-driven, and thus essentially declarative and generative, approach to address them: the user should only declaratively specify the non-functional "contract" that a system component shall satisfy in the provision of its functional services; a transformation engine would then associate proven solutions to implementation of the contractual needs and automatically deploy them in the system product.

Property Preservation. In keeping with the general principles of MDE, ASSERT adopted a multi-staged production process that proceeds from a user-level specification across a number of automated transformations, to eventually producing source code for the application, with tailored bindings to the execution environment. On account of the high-integrity nature of ASSERT applications, each stage of transformation must provably exhibit the capability of capturing all of the properties attached to the (more abstract) input model as well as of fully preserving them in the (increasingly more concrete) output model.

Later in this paper we shall elaborate more on the properties of interest, their origin and the attainment of guarantees of preservation.

Timing Predictability. Timing predictability is one of the two key non-functional concerns that were addressed in ASSERT. A system is timing predictable if quantitative reasoning about its timing behavior at run time can be (economically) made off line [15]. The attainable extent of timing predictability in a system is determined by the nature and capabilities of the mechanisms and policies availed to govern the scheduling of software execution over time. ASSERT warrants timing predictability by:

1) providing and exploiting definite knowledge about the nature and semantics of the entities that are allowed to exist at run time;
2) exclusively adopting concurrency-control policies whose effect can be statically and accurately analyzed;
3) providing and exploiting run-time mechanisms that are fully deterministic in time and semantic effects, and that help ensure the preservation of statically stipulated timing properties.

In the literature of real-time computing, those three dimensions are collectively addressed by the subsuming notion of "computational model".

Isolation. The ASSERT approach provides isolation with respect to time, space and communication. This threefold form of isolation is obtained by way of partitioning. A software system in ASSERT is made up of an collection

of logical partitions deployed on locally distributed physical nodes. Partitions permit to contain local faults and stop them from propagating outside. Local faults manifest themselves in a partition as a violation of contractual stipulations with respect to: timing budget, which cannot be exceeded; memory access, which cannot reach outside a statically allocated memory region; and communication bandwidth, which cannot exceed the quota assigned over time periods.

Distribution Transparency. This property implies that software components need not know and make no assumptions about their location in the system topology. The topology-dependent transport of all communications is transparently taken care of by the coordinated action of the transformation process and a dedicated middleware layer.

3. The ASSERT approach

ASSERT treats non-functional concerns using a declarative and generative approach in keeping with the model-driven development style. The user attaches non-functional "contracts" to functional interfaces. The user designs the latter using specialized languages and tools, which may or may not include their own automated code generation capabilities and thus may encompass implementation. The user instead only *declares* the former and leaves it to the ASSERT infrastructure to capture the non-functional requirements that correspond to the interface contracts and to realize them by composition of proven components, patterns and archetypes.

Once implemented, non-functional contracts endow non-functional properties on system components (e.g.: non-exceedable thresholds on execution time and communication bandwidth; a level of privilege or criticality; a fenced memory region). The satisfaction of some of those properties can be asserted statically, by the very specification of the applied transformations. For some others instead run-time enforcement is needed: this paper traces the way in which the relevant properties are preserved throughout the transformation process and illustrates the way in which the execution environment actively contributes to this goal. Owing to this distinguishing characteristic, the execution environment required for ASSERT applications is called *Virtual Machine*.

The ASSERT approach may be regarded as a specific incarnation of the MDE paradigm, which elevates the Model Driven Architecture [16] initiative to a full-blown engineering discipline. One of the essential characteristics of model-driven engineering is to promote rigid separation between platform independent aspects (that are described in a "Platform Independent Model", PIM) and platform specific ones (that are described in a "Platform Specific Model", PSM). The user operates in the PIM space and leaves the realization of the corresponding PSM to a suite of automated transformations. In ASSERT the PIM space includes full functional modeling and the declarative specification of non-functional contracts, while the PIM to PSM transformations have the additional quality of being provably property preserving. The underlying motive in favor of this approach is its economy of scale: the transformations, which are platform specific, need to be realized and proven only once per platform, regardless of the number of systems that may be realized on that platform.

3.1. Achieving separation of concerns

ASSERT achieves separation of concerns by using a specific notion of *non-functional container*. ASSERT containers encapsulate the functional code that results from the user specification and dote it with trustworthy (i.e., proven) pre-defined implementation of the non-functional contract attached to it. This encapsulation occurs transparently to the functional specification, which is thus neither perturbed nor affected, so that any functional properties proved on it at model level continue to hold true during execution.

For any given platform there exist a finite set of trustworthy implementations of non-functional contracts. All that automated transformation needs is to relate a contract to a predefined implementation, which was proven correct beforehand, and configure and deploy it for the specific use.

The ASSERT notion of container bears considerable resemblance to the CORBA Component Model (CCM, [17]).

The rationale behind the CCM was the wish to reuse solutions that solve recurrent infrastructural problems and relieve the user from the need to create ad hoc solutions for problems which only vary for non-functional characteristics. The essence of containers thus is to provide predefined and expandable solutions to the transparent provision and handling of non-functional services.

A CCM container provides an execution environment for one or more *components* (where a component is "the implementation entities that export a set of interfaces usable by conventional CORBA clients as well as other components", [17]) and permits them to use a number of predefined solutions to address persistence, transactions, protection, etc. (For details on the notion of component software, the reader is referred to [8].)

3.2. The ASSERT containers

ASSERT containers are modeled after the CCM paradigm, but with special attention to the properties listed in Table 1. The classes that satisfy a cohesively related set of functional requirements is embedded into a component structure, whose implementation view is called OPCS (for OPeration Control Structure). The OPCS is then encapsulated into a container that adds non-functional capabilities to the functional contents that the OPCS realizes.

ASSERT defines two types of containers. The containers that exist in the PIM space, which are called *APplication Level container* (APLC), enable the user to attach non-functional contracts to the interfaces of functional components. The containers that exist at PSM level instead, which are called *Virtual Machine Level Container* (VMLC), represent the implementation-level solution to the user specification of non-functional contracts.

Figure 1 provides a simplified diagrammatic representation of the ASSERT container structure and taxonomy.
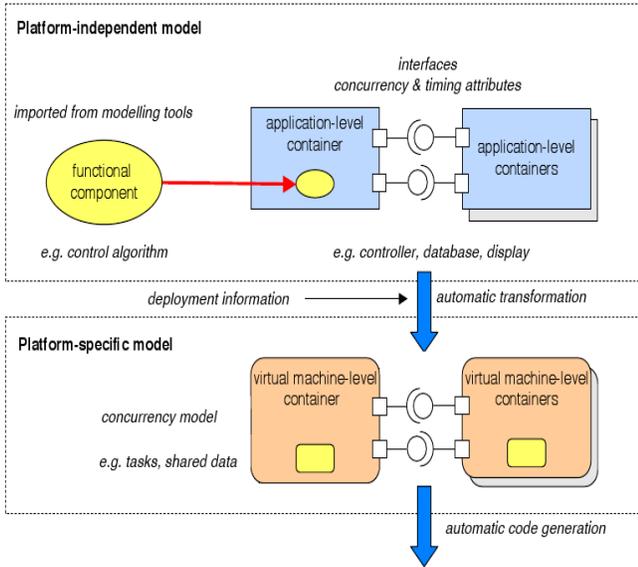

Figure 1. Components and containers in ASSERT.

3.3. The PIM space

The PIM space in ASSERT encompasses three views: functional; interface; and deployment. Each such view addresses a distinct and complementary aspect of the system model. The functional view contains the functional design of application components: this is where the user's OPCS are specified. The interface view embeds functional components in APLC and decorates the provided interface (PI) and required interface (RI) of the embedded components with the desired non-functional contracts. The deployment view specifies the assignment of APLC to logical partitions and the mapping of logical partitions to physical nodes.

3.4. The PSM space

The non-functional contracts set on the PI and RI of APLC in the PIM space are realized by lower-level containers (the VMLC). Such VMLC are the sole legal entities that may populate the PSM space. The generative process that transforms individual APLC into aggregates of VMLC

is formally specified in [18]. The intent of that specification is to prove the completeness and the property-preserving nature of the transformations. Completeness is needed to ensure that all components that are expressed in the PIM space can definitely be transformed into legal entities in the PSM space: this is one of the essential premises of the correctness by construction property of ASSERT systems. Property preservation is needed instead to be able to claim that all the properties endowed on PIM components are fully transferred to and correctly realized in the corresponding PSM components.

VMLC are realized by the composition of predefined behavioral archetypes determined in full accord with the adopted computational model chosen to attain timing predictability. In ASSERT the computation model of choice is the Ravenscar Computational Model (RCM). The RCM derives, in language neutral terms, from the Ravenscar profile of Ada [19]. The RCM is especially valuable in that all systems that comply with it are by definition: (i) amenable to static analysis for time, space and other dimensions of behavior, since they abide by a deterministic model of concurrent execution; and (ii) can also be run on small and efficient run-time kernels that may be economically certified.

The Ravenscar profile places restrictions on the PSM space to ensure that all models in that space be statically analyzable. The Ravenscar profile: (i) forbids the use of language constructs that may incur non-determism or unbounded execution time; (ii) allows only asynchronous one-way communications mediated by shared resources equipped with a deterministic synchronization protocols [20], [21] so that direct inter-task communication are prohibited; (iii) requires that threads to have a single suspension and a single (cyclic or sporadic) source of activation events.

4. The ASSERT Virtual Machine

A naïf approach to the development of high-integrity systems devotes considerable effort to performing verification by static analysis, but omits to ensure that the resulting stipulations shall not be broken at run time.

Numerous analysis techniques often require the user to specify worst-case assumptions in order to assess the feasibility of the input model in the dimension of interest. Now, if the user model is proven feasible against the given assumptions (also considering how difficult they often are to determine in a non-overly pessimistic manner) it would seem to be vital for the user that those assumptions could never be violated during execution.

Surprisingly however, those assumptions often are completely forgotten about when the executable application is produced. Until the recent past this oblivion was forced on the user by the inability of the programming language and execution environment to "understand" the notion of assumption, as a key element of a contractual obligation to
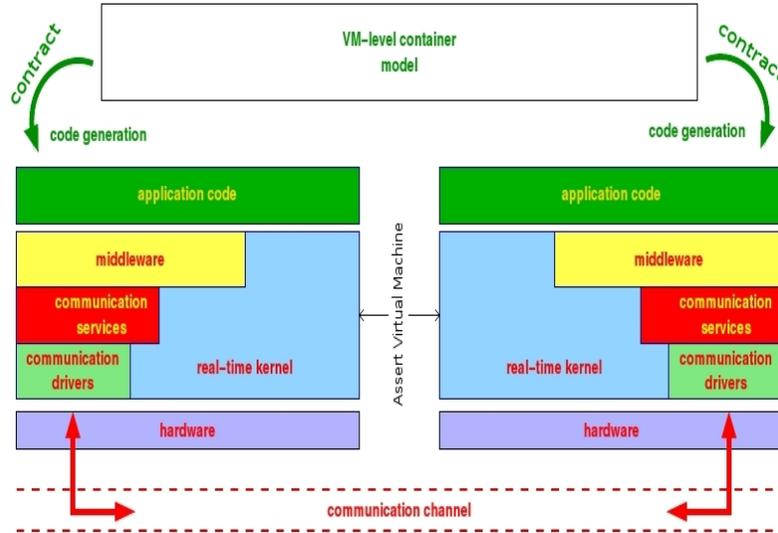
Figure 2. Architecture of an ASSERT system.

maintain. Technologies begin to emerge however that fare considerably better in that regard and that permit to construct an executable, and an execution environment alike, more acutely aware of their respective obligations.

The ASSERT vision in this respect was to specify and realize an execution environment capable of understanding the non-functional contracts exposed by the run-time components (VMLC) and deploy run-time mechanisms to ensure their preservation and prevent the effects of violations from propagating beyond the failing component.

As we said earlier on, the execution environment retained for ASSERT applications is called the ASSERT Virtual Machine (AVM). The AVM accepts for execution no other run-time entities than VMLC The presence of violating entities in the application code submitted for cross compilation to the AVM is captured by the compiler and the generation of the executable is failed.

In order for this provision to be effectively implemented it is important that as many contract-enforcement features as possible are directly realized by explicit language-level commands whose correctness can be guaranteed by the compilation system. This is the case in ASSERT, where there is tight integration between the AVM and its cross compiler. The programming language of choice is Ada [22], which has a wealth of such features, in addition to a standard thus enforceable specification of the Ravenscar profile.

The AVM is comprised of three distinct, complementary and integrated layers, as shown in Figure 2.

- *Real-Time Kernel*
- *Middleware*
- *Communication services.*

An instance of the AVM is deployed on every single processing node of an ASSERT system.

4.1. The Kernel layer

This component implements the Ada incarnation of the Ravenscar Computational Model. The RCM specifies the concurrency model which all run-time entities must comply with both statically and dynamically. The target cross compiler (which was developed for ASSERT by the Universidad Politecnica de Madrid [23] on the basis of AdaCore's GNAT [24]) includes the run-time libraries that form the Kernel layer and links them with the application software at all places where required by the source code automatically generated by the PSM-to-code transformation. It is important to notice in this respect that no concurrent semantics can result from the code produced from functional models. For this reason, the ASSERT development process may accommodate the use of code generation engines with functional models as long as they are designed in compliance with the ASSERT principles.

The Kernel layer includes facilities for run-time enforcement of the parts of non-functional contracts that concern time. If adequate hardware support was available with the target processor retained in the ASSERT project (the Leon processor [25]) the Kernel layer should also take care of the space-related part of non-functional contracts: this capability is not implemented yet in the current release of the AVM.

4.2. The Middleware layer

This component provides VMLC with distribution transparency in order that application components need not bother

about the logical and physical topology of the system. All aspects that concern name resolution, routing and transport are taken care of transparently to the application.

This component is realized by PolyORB-HI, a high-integrity derivative of the PolyORB schizofrenic middleware developed by Get/Telecom Paris [26]. The Middleware layer complies by construction with the RCM. In fact, the whole Middleware layer may be regarded as an application running on the AVM Kernel: as such, the Middleware may be generated in exactly the same manner as the user application is and with the same guarantees of property preservation that can be assured by the AVM Kernel.

4.3. The Communication layer

This component takes care of the physical communication over the interconnection medium. In the ASSERT-project release of the AVM this component was realized by the union of two medium-specific subcomponents, both developed by SciSys in the UK. Their role is not crucial to the focus of this paper and therefore they are not discussed further.

5. Coverage of System Properties

Table 2 summarizes how the preservation measures of system properties are apportioned to the layers of the AVM.

Property to preserve	Responsible AVM layer
Timing predictability	Kernel
Temporal Isolation	Kernel
Spatial Isolation	Kernel (not implemented yet)
Communication Isolation	Communication services
Distribution Transparency	Middleware

Table 2. AVM coverage of system properties.

The coverage of the time-related properties shall be discussed in the remainder of this section while the other dimensions are left to Section 6.

Timing predictability. The very definition of the RCM and its implementation in the AVM do guarantee the level of timing predictability needed to for static analysis. Details on the run-time semantics of the RCM are extensively given in [27]. Details on the model-based static analysis supported in ASSERT are given in [28].

Temporal isolation. The guarantee of temporal isolation is attained in ASSERT by the combined operation of two enforcement mechanisms: (i) no thread attached to the run-time representation of the VMLC produced by transformation of the PIM may violate the worst-case execution time (WCET) stipulated for feasibility analysis; (ii) no logical partition, which results from the aggregation of VMLC (and thus threads) in a thread group known to the AVM Kernel, may consume execution time in excess of a maximum value stipulated for feasibility analysis.

Mechanism (i) consists in attaching an execution-time timer to each thread, initialized to the WCET value stipulated in the relevant contract, and an event handler that is triggered when a violation occurs. The timer, which is primitively implemented in the AVM Kernel, monitors the CPU time consumed by the thread between an activation and the subsequent suspension. The handler, which exists as a predefined archetype in the PSM space, is attached to the relevant thread by the PIM transformation process using a user-defined violation-handler operation in the instantiation.

While [29] shows that mechanism (i) is actually sufficient to warrant time isolation, mechanism (ii) permits to provide useful flexibility, in the form of user-defined slack times, to partition-wide fluctuations in WCET violations: when a thread incurs a WCET violation an extra slack time may be granted for it to complete execution, which will however be detracted from the time budget of the relevant partition in order that no other partitions should suffer from violations occurring outside of them.

The AVM Kernel implements a hierarchical scheduling policy based on fixed-priority preemption (FPP) in the priority-band based way supported by the Ada standard [22]. A global FPP scheduler assigns the processor to the partition in the highest band in which there is a ready thread. The local scheduler, which may use a score of policies other than FPP, dispatches one of its thread in accord with the policy in force in it.

Table 3 provides a fine-grained view of the AVM Kernel coverage of the contract clauses that reflect time-related system properties to preserve at run time. (Contract clauses are 'explict' if set by the user in the PIM space and 'explicit' if set by the transformation process.)

Property	Contract clause	Specification
Timing Predictability	Thread priority	Implicit
	Thread deadline	Explicit
	Thread WCET	Explicit
	Thread period or MIAT	Explicit
	Periodic/sporadic release	Implicit
	Deadlock free synchronization	Implicit
	Predictable scheduling	Implicit
Temporal Isolation	Execution-time timers	Implicit
	Group budgeting	Implicit

Table 3. Coverage of time-related contract clauses.

6. System Production Process

The production process of an ASSERT system starts from the PIM designed by the user. The ASSERT infrastructure includes a graphical editor realized as an Eclipse plug-in, in which the user model is described using an ASSERT-specific profile of UML [30].

Two stages of automated model transformation ensue from the commitment of the user model in the PIM space: first, the transformation of APLC into-VMLC (A2V-T) and

then the transformation of VMLC into source code (V2C-T). Let us briefly review them in some coarse detail.

6.1. Overview of the transformation

6.1.1. From Functional view to Interface view.
In this view the user designs the components that are to satisfy functional the requirements set on the system. Modeling in this view may be assimilated to drawing class diagrams and composite structure diagrams in classic UML. The only important difference is that the modeling language supported by the editor actively prevents the expression of semantics that reach into non-functional properties.

Once functional components are defined, the user moves on the Interface view to aggregate them into APLC and to decorate their interfaces with non-functional contracts.

6.1.2. Deployment view.
In this view the user specifies the physical architecture of the system and consequently determines the constraints on physical resources (time, space and communication) that must be respected by the application components. This view is expressed in terms of processing nodes and interconnections. For each processing node the following specification information must be supplied:

- *processor*: type and clock frequency
- *memory*: size, access time and bandwidth
- *virtual machine*: available priority range for threads and interrupts, global scheduling policy, time overheads of all kernel services.

The system specification includes the interconnection between processing nodes, with bandwidth capacity, maximum and minimum packet size, transmission time per packet, transmission type, and protocol.

Once the physical system is specified, the user moves on to define the logical partitions that must be attached to the physical nodes of the system. A logical partition is defined as an aggregation of APLC assigned to a given physical node. For each logical partition, the instance view specifies:

- the processing node where the partition is to reside
- the criticality level assigned to the partition
- the local scheduling policy to apply to the partition
- the storage budget assigned to the partition
- the execution time budget assigned to the thread group in the partition.

Once APLC are deployed on the partition of residence, the association between the PI of an APLC and the matching RI of another APLC create interconnects between the corresponding processing nodes. Each resulting interconnect must be attached to a non-functional contract that specifies the worst-case needs of the inter-APLC communication on that interconnect, expressed in bytes per period of time. Those contract clauses address the communication isolation properties (cf. Table 2) which are passed on to the Communication services for enforcement at run time.

In this manner, the user is granted full distribution transparency and no involvement whatsoever in the generation of all of the required stubs and skeletons and in the data insertions in the Middleware level routing tables that provide the transport services. This completes the coverage of the system-level properties listed in Table 2.

6.1.3. Transformation algorithm.
This transformation operates in two subsequent stages: (i) the first stage turns each APLC instance into a set of VMLC, and therefore specifies the run-time entities that shall execute on the AVM on behalf of the originating APLC; (ii) the second stage detects which associations between APLC the deployment specification has caused to become remote and then creates all of the infrastructure needed for them to be transparently distributed as well as to ensure the preservation of the non-functional contract clauses related to the isolation of their communications.

Overall, the A2V-T performs the following activities:

1) Detection of remote communications and transparent realization of stubs aggregated in node-local APLC.
2) Transformation of APLC into VMLC.
3) Transparent insertion of skeletons next to VMLC that expose remote interfaces.
4) Definition and insertion of VMLC that represent the node-local instantiation of the Middleware layer.
5) Cascading, calculation and setting of all non-functional contract clauses that descend from system properties to be validated by static analysis and to be preserved at run time.

The product of this transformation is a PSM-level specification called, in ASSERT, the *Concurrency View*.

The subsequent transformation turns the VMLC specified in the Concurrency View into source code for cross compilation with the Kernel and for execution on the target platform. In ASSERT this process was extremely facilitated by the choice of Ada as the target programming language and the use of Ada in the implementation of the AVM itself. Ongoing studies run by the author's team are investigating the suitability and performance of the Real-Time Specification for Java for the same purpose.

7. Conclusions

This paper has reported on an MDE approach for the production of high-integrity software systems that warrants the preservation of non-functional properties from the user model to the execution environment.

A continuation project, named CHESS, has recently been launched to extend the ASSERT capability of property preservation to the provision of compositionality guarantees in the application space. The goal in CHESS is to enable the user to develop PIM-level components, to endow them with

functional and non-functional properties proven in isolation, and to be granted that they will be preserved in the assembly with heterogeneous components as well as during execution.

References

[1] Office of Government Commerce (UK), "The IT Infrastructure Library (ITIL)," http://www.ogc.gov.uk/guidance_itil.asp.

[2] I.-R. Chen and B. Cukic, "High assurance software systems," *Comput. J.*, vol. 49, no. 5, pp. 507–508, 2006.

[3] B. Selic, "From Model-Driven Development to Model-Driven Engineering," Keynote talk at ECRTS'07, http://feanor.sssup.it/ecrts07/keynotes/k1-selic.pdf.

[4] ——, "Personal reflections on automation, programming culture, and model-based software engineering," *Autom. Softw. Eng.*, vol. 15, no. 3-4, pp. 379–391, 2008.

[5] Automated proof-based System and Software Engineering for Real-Time systems, http://www.assert-project.net.

[6] ARTEMIS Joint Undertaking, "Embedded Computing Systems Initiative," http://www.artemis-ju.eu.

[7] E. Dijkstra, "On the role of scientific thought," in *Selected Writings on Computing: A Personal Perspective*. Springer-Verlag New York, Inc., 1982, pp. 60–66.

[8] C. Szyperski and al., *Component Software: Beyond Object-Oriented Programming*, 2nd ed. Pearson, November 2002, ISBN-13: 9780201745726.

[9] R. Bruni, A. Lluch Lafuente, U. Montanari, and E. Tuosto, "Style Based Reconfigurations of Software Architectures," Università di Pisa, Tech. Rep. TR-07-17, 2007.

[10] A. Easwaran, I. Shin, O. Sokolsky, and I. Lee, "Incremental Schedulability Analysis of Hierarchical Real-Time Components," in *Proc. EMSOFT 2006*, 2006, pp. 272–281.

[11] S. Bliudze and J. Sifakis, "The Algebra of Connectors - Structuring Interaction in BIP," in *Int. Conf. EMSOFT*, 2007, pp. 11–20.

[12] D. Schmidt, "Model-driven engineering," *IEEE Computer*, pp. 25–31, February 2006.

[13] A. Sangiovanni-Vincentelli and R. Passerone, "Contract-based formalisms for heterogeneous and hybrid systems," http://www.artist-embedded.org/Foundations-of-Component-based.html, 2007.

[14] T. Vardanega, "A Property-Preserving Reuse-Geared Approach to Model-Driven Development," in 12th *IEEE Int. Conf. on Embedded and Real-Time Computing Systems and Applications*. IEEE, August 2006, pp. 223–230.

[15] D. Jensen, "Real-Time for the Real World," http://www.real-time.org/realtime.htm.

[16] The Object Management Group (OMG), "MDA guide version 1.0.1," http://www.omg.org/mda.

[17] G. Edwards, G. Deng, M. Xiong, and A. Gokhale, "Evaluating real-time publish/subscribe service integration approaches in qos-enabled component middleware."

[18] D. Cancilia, T. Vardanega, I. Hamid, and E. Najm, "An HRT-UML/RCM Interface Grammar for AP-level Modeling," University of Padua, ASSERT Technical Note, October 2006.

[19] A. Burns, B. Dobbing, and T. Vardanega, "Guide for the Use of the Ada Ravenscar Profile in High Integrity Syst ems," *Technical Report YCS-2003-348, University of York*, 2003.

[20] J. B. Goodenough and L. Sha, "The priority ceiling protocol: a method for minimizing the blocking o f high priority Ada tasks," in *Proc. of the 2nd International Workshop on Real-time Ada Issues*, 1988, pp. 20–31.

[21] T. P. Baker, "Stack-based Scheduling for Realtime Processes," *Real-Time Systems*, vol. 3, no. 1, pp. 67–99, 1991.

[22] ISO, "Ada Reference Manual. Language and Standard Libraries. Consolidated Standard ISO/IEC 8652:1995(E) with Technical Corrigendum 1 and Amendment 1," 2005.

[23] Universidad Politécnica de Madrid, "GNATforLEON cross-compilation system," http://polaris.dit.upm.es/ ork.

[24] AdaCore, "GNAT Pro," http://www.adacore.com/home/gnatpro/.

[25] *LEON2 Processor User's Manual*, Aeroflex Gaisler, 2005, http://www.gaisler.com/cms.

[26] B. Zalila, J. Hugues, and L. Pautet, "PolyORB-HI," http://aadl.enst.fr/polyorb-hi/.

[27] T. Vardanega, J. Zamorano, and J. A. de la Puente, "On the Dynamic Semantics and the Timing Behavior of Ravenscar Kernels ," *Real-Time Systems*, vol. 29, pp. 59–89, 2005.

[28] M. Bordin, M. Panunzio, and T. Vardanega, "Fitting Schedulability Analysis Theory into Model-Driven Engineering," in *Proc. of the 20th Euromicro Conference on Real-Time Systems*, 2-4 July 2008, pp. 135–144.

[29] J. A. Pulido, "Arquitectura de software para sistemas de tiempo real particionad os," Ph.D. dissertation, Universidad Politécnica de Madrid, July 2007.

[30] S. Puri, M. Trevisan, and T. Preuss, "HRT-UML2 - The HRT-UML/RCM Toolset - User Manual," Intecs, ASSERT Technical Note, November 2007.