Proof-Based System Engineering Using a Virtual System Model

Martin Biely[1], Gérard Le Lann[2], and Ulrich Schmid[1]

[1] Technische Universität Wien, Embedded Computing Systems Group E182/2
Treitlstraße 3, A-1040 Vienna, Austria
{biely,s}@ecs.tuwien.ac.at
[2] INRIA Rocquencourt, Project Novaltis
Domaine de Voluceau BP 105, F-78153 Le Chesnay Cedex, France
Gerard.Le_Lann@inria.fr

Abstract. This paper provides an overview of Proof-Based System Engineering (PBSE), which aims at improving the current practice of developing computer-based systems. PBSE is of particular relevance for safety critical applications and other systems where dependability properties are essential. This is particularly the case for applications in the aerospace domain targeted in the EC FP6 Integrated Project ASSERT. Applying PBSE both permits to eliminate most common design faults before embarking on the development of a system and maximizes reuse, which leads to significant savings in time and budgets. Particular emphasis is put on the requirements capture phase of PBSE, where a virtual system model is used as a novel means to structure the information to be captured.

1 Introduction

Stringent requirements for high availability, high reliability and safety in mission- or/and life-critical applications entail specific and complex constraints on the design, verification and validation of *computer-based systems* (CBS). The challenges thus involved are addressed by the *Proof-Based System Engineering* (PBSE) method, which builds upon INRIA's TRDF method ("Traitement Distribué", "Temps Réel", "Tolérance aux Fautes"), a generic method that has already been applied successfully in a number of former projects [1,2]. PBSE is currently applied in the FP6 Integrated Project ASSERT.[1]

Unlike most software engineering approaches, PBSE targets the entire CBS of an application, not just the software part of its constituents' embedded systems. Examples are the worldwide distributed CBS for a bank or — in the case of ASSERT — the CBS that spans spacecraft, the International Space Station, and

[1] ASSERT (IST-004033) is an IST-FP6 Integrated Project sponsored by the European Commission under the strategic objective of "Embedded Systems". Coordinated by the European Space Agency (ESA), the consortium consists of 29 partners from both academia and industry. Consult www.assert-online.org for further details.

ground stations. Traditional formal/informal software engineering methods are primarily concerned with *how to build the specification right*, i.e., how to correctly implement some given specification. PBSE is orthogonal to these methods as it addresses the issues involved with *how to build the right specification*, which consists of:

– *building an adequate specification of the problem(s) to be solved*, by mandating a dedicated requirements capture phase prior to any system design, validation and implementation work,
– *building a correct specification of the solution(s)*, with a priori and maximum reusability of efforts, by mandating "forward" proofs in every step of the solution design, rather than "backwards" verification and testing.

PBSE focuses entirely on the CBS-centric non-functional requirements "hidden" in an application, however. It thus actually allows to separate functional requirements (application semantics) from non-functional requirements [3]: Application programmers, who may use standard formal/informal software engineering methods[2], can safely ignore non-functional aspects during functional analysis and design. PBSE experts, on the other hand, can abstract away functional requirements in the course of their work, which rests upon splitting the non-functional requirements into a set of *models* that specifies assumptions about the CBS's environment, and a set of *properties* that specifies desired system-level services and their QoS. The system-level solutions developed according to PBSE principles will guarantee that the CBS satisfies those properties in any environment that matches the assumptions stated in the models.

One of the primary purposes of PBSE is to eliminate faults made in the early phases of the overall life cycle: It is well known that faults made in the course of requirements capture phases are the dominant causes of project setbacks or operational failures, hence the major contributors to inflated costs and project overruns. Another primary purpose of PBSE is to reduce the complexity of the system integration and final testing phases, phases which are not well mastered under current practice. Finally, PBSE aims at composability checking, targeting the reuse and composition of designs and proofs, not just the reuse and composition of software or hardware components.

This paper provides an overview of the rationale and life cycle of PBSE, and introduces the virtual system model as our primary means to structure the requirements capture phase. It is organized as follows: The rationale for the need of a proof-based approach and some related work is given in Section 2. An overview of the PBSE life cycle, with particular emphasis on the PBSE requirements capture phase, is contained in Section 3. Section 4 provides a short example of the reuse possible with PBSE. The definition and usage of the virtual system model is presented in Section 5. A concluding discussion of PBSE in Section 6 completes the paper.

[2] Using formal SW engineering methods puts you in the desirable situation of having a continuous chain of proofs from problem specification to implemented solution.

2 Why PBSE ?

Consider critical systems, where criticality is related to the possible loss of life, mission or simply money. Obviously, such systems should be designed in a way that prevents such losses, or, more realistically, makes them sufficiently unlikely. For air traffic control systems, for example, it is required that system unavailability shall be less than 3 seconds a year, which translates into an availability figure of $1 - 10^{-7}$. With today's practice, however, too many of these systems fail, and too many projects are canceled, are late or more expensive than planned due to the difficulty of meeting such stringent requirements.

Software design & software development is commonly blamed for these problems, giving raise to the so-called "software crisis" in critical systems design. And indeed, as software became the dominating factor in today's computer-based systems, there is always some piece of software running when a system failure occurs. However, simply accusing software turns out to be wrong or at least misleading in many cases, since doing this ignores the difference between the cause of a system failure, i.e. the fault, and its observed manifestation, i.e. the failure.

In fact, several studies show that software is better than its reputation: For instance, an analysis of the causes of failures of the US public switched telephone network [4] shows *"that SW errors caused less downtime (2%) than any other source of failure except vandalism"*. Rather, overloads were recognized as the dominant cause (44%). Another example where software was blamed for a system failure is the well-known loss of the Ariane 5/501 launcher, which caused a financial loss in the order of 450 M€ and a 1 year delay for the Ariane 5 program. Although the inquiry board [5] concluded that poor software engineering practice was the culprit, other problems actually caused the failure [6,7].

Rather than in software [engineering], these and many other critical system failures have their roots in poor system engineering practice [8]. Of course this is not meant to suggest that the computer industry does not have problems in the field of software engineering, but rather that there are other areas (i.e., system engineering) that are even less mastered and have not received enough attention.

One major reason of failure is related to the specification generation process: With formal software engineering methods, under some restrictions, it can be verified that specifications are implemented correctly. But where do these specifications come from? It does not help much to be provided with a software component "proved correct" vis-à-vis its specification, if that specification is inappropriate ("incorrect") for the application/system problem considered. Proper requirements engineering methods [9,10,11,12] must be utilized to provide an agreed-upon specification of the problem to be solved. In general, however, this is difficult due to the inevitable intertwining of requirements and solutions [13] and the often conflicting requirements of different stakeholders [14]. In the context of ASSERT, the problem is further exacerbated by the difficult fault-tolerant distributed real-time computing problems typical of aerospace applications.

Another major problem is the level of complexity involved with proving systems-in-the-large [3,15]: Even a locally verified system component can suffer from inconsistencies and hidden non-functional dependencies with respect to other components in the system. So if such components, which behave correctly when run in isolation from each other, are executed together within a system, they could suffer from undesired interference and hence fail. The resulting failure is observed in the execution of the software, but the actual fault is rooted within poor system engineering practice: Global verification would have spotted system-level inconsistencies and hidden non-functional dependencies. Unfortunately, however, such techniques suffer from well-known state explosion problems and are hence infeasible for most real-world-size problems. Moreover, they are necessarily “a posteriori” verification approaches, which do not a allow the development of solutions that are correct-by-construction.

PBSE is the only method we are aware of that addresses these challenges in a common framework: PBSE/TRDF shares some of the goals of the Design-by-Contract approach [16] and the B method [17], notably the mandated use of non-ambiguous specifications and the fulfillment of proof obligations. However, PBSE addresses system-level concerns, regardless of the implementation technology resorted to in fine, rather than software-related concerns only. In the remainder of this paper, we will try to shed some light on how this is accomplished.

3 The PBSE Life Cycle

Before giving an overview of the phases of the PBSE life cycle, we need to introduce some basic notations. As mentioned above PBSE is concerned with building the correct specification of the problem to be solved, as well as building the correct specification of the solution. A *specification of a problem* will be denoted $\langle \mathrm{Z} \rangle$, with $\langle \mathrm{z} \rangle$ denoting the set of unvalued variables in $\langle \mathrm{Z} \rangle$. The *Design specification of a system solution* will be denoted [S], with the set of unvalued solution variables [s] that correspond to the unvalued problem variables $\langle \mathrm{z} \rangle$. Typical examples of such unvalued variables are process sets, deadlines, worst-case execution times, invariants for logical safety, density of failure occurrences. Note that the size and type of $\langle \mathrm{z} \rangle$ and [s] reflect the genericity of the specification of the problem $\langle \mathrm{Z} \rangle$ and the solution [S], respectively. The design specification [S] is referred to as specification of a solution, because its implementation is the solution, denoted S, of the problem stated in $\langle \mathrm{Z} \rangle$. In ASSERT we do not consider a specific mission, but rather (two) families of missions, resulting in two very generic pairs $\{\langle \mathrm{Z} \rangle$, [S]$\}$ of problem and corresponding solution specification, which are referred to as *System families (SF)*.

A problem specification $\langle \mathrm{Z} \rangle$ actually comprises two sub-specifications:

– *Models* $\langle \mathrm{m.Z} \rangle$, which stipulate operational, technological, and environmental assumptions. They specify the *adversary* (Adv) for (the designers of) [S].
– *Properties* $\langle \mathrm{p.Z} \rangle$, which stipulate the desired services and QoS. They must be guaranteed by the operational system S (assuming [S] is implemented correctly) in the presence of an adversary no stronger than $\langle \mathrm{m.Z} \rangle$.

Specifications such as $\langle \mathrm{Z} \rangle$ are written in restricted natural language: All terms in $\langle \mathrm{Z} \rangle$ must have formal or technical definitions in scientific or engineering dis-

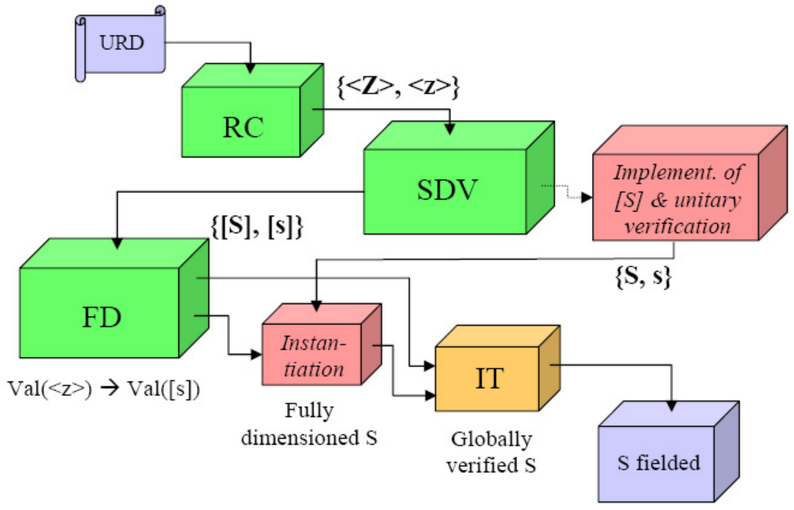

Fig. 1. *Schematic representation of the entire PBSE life cycle*

ciplines (computer science terminology, mathematics, etc.), to the exception of conjunctions, articles, and other syntactic elements. Examples include:

– “distributed” ≡ “current global state cannot be known”,
– “serializable execution” ≡ “interleaved execution identical to some sequential execution”,
– “Byzantine” ≡ “arbitrary behavior”.

Figure 1 shows a schematic representation of the entire PBSE life cycle. Phases that are proper to PBSE are the RC, SDV, FD and IT phases. The RC and SDV phases precede the instantiation of [S], the FD phase precedes the instantiation of every specific customized release of S, and the IT phase serves to derive automatically the suite of tests needed to conduct the integration testing (global verification) of S. The implementation of [S] and unitary verification, on the other hand, are fully within the realm of formal/informal software engineering.

Therefore, the PBSE process spans all RC, SDV, FD and IT life cycle phases whenever a novel problem Z is considered and some solution S is to be fielded. Conversely, after a pair $\{\langle Z\rangle, [S]\}$ has been constructed, only the FD and the IT phases need be conducted for the fielding of some specific release of S. Customized releases are obtained by assigning values to free problem variables in $\langle z\rangle$ and running the FD phase, which produces values for the free system variables in [s].

3.1 The Requirements Capture Phase

The *Requirement Capture* (RC) phase bridges the gap between the application-centric requirements and the resulting CBS-centric requirements. The input of the RC phase is a document, called *user requirements document* (URD) in the sequel, which describes the objectives of the application in the client's domain-specific terminology. The result of the RC phase is a specification of the system-level computer-based problems/requirements $\langle Z \rangle$, which matches the URD.

In the RC phase, the application/mission requirements are separated from reuse considerations — PBSE accommodates the mandatory use of pre-existing partial solutions (e.g., COTS products). Design concerns, related to how $\langle Z \rangle$ could be solved and/or some [S] implemented, are totally ignored.

An existing or novel component of a to-be-designed system is called an *entity*. The modeling of entities is done similar to the I/O automata formalism [18]:

– Inputs are (specifications of) incoming events and associated shared data, arrival laws (loads), failures,
– internals are (specifications of) processes (structure, worst-case execution times) and shared data/states,
– outputs are (specifications of) outgoing events and associated shared data, failures.

These models are intrinsic to a given entity. Inputs and outputs correspond to behaviors in I/O automata. Properties serve to specify desired properties, which may differ from (intrinsic) outputs.

Operationally, requirements capture is a two step process. In step 1, models and properties are captured on a per entity/level basis, in strict isolation of each other. This work can be done by multiple teams in parallel, for different entities or collections of entities. Since the collective behavior of sets of entities (e.g. multiple programs multiplexed over a CBS) is usually also relevant, the desired properties for such sets may also be captured. An example would be the "serializability" property [19] for a set of application programs that share updatable and persistent data. Finally, since computer-based systems are never built from scratch in real projects, it is possible to specify, at RC time, which pre-existing components (hardware, software) are to be reused. To be part of the proof chain that spans from $\langle Z \rangle$ to [S], however, a reused component E must have a companion technical leaflet (see Section 3.2) that also includes its $\langle Z(E) \rangle$.

In the second step of the RC phase, every entity E is revisited, considering all models captured at the end of step 1 which are appropriate. For example, some failure models and failure occurrence models have been captured (during step 1) for processing entities (abstractions of processors). Causes of such failures are cosmic rays, vibrations, and so on. Separately, some failure models and failure occurrence models have been captured (during step 1) for application entity E (abstraction of a software/functional process). Causes of such failures are software design and implementation faults. During step 2, entity E is "revisited", in order to specify its intrinsic behaviors in the presence of failing processing entities (ignored at step 1).

In the absence of tools[3], the RC phase is typically performed via interactive meetings, where the stakeholders scan the application's URD according to the PBSE *RC Guide*. The RC Guide is a menu of classes of models and of properties orthogonal to each other, that constitute a multidimensional space Π. The 10 classes that define Π (6 model classes, 4 property classes) are as follows:

– Computational Models, Resource and Data Models, Process Models, Event and Event Arrival Models, Failure Models, Failure Occurrence Models
– Logical Safety, Liveness, Timeliness, Dependability properties

Any specification $\langle \mathrm{Z} \rangle$ corresponds to a region within Π.

It is this process that makes it possible to capture the properties $\langle \mathrm{p.Z} \rangle$ and the adversary $\langle \mathrm{m.Z} \rangle$ for the entire CBS. Thanks to $\langle \mathrm{Z} \rangle$, it is then possible to detect some impossibility results at the RC stage (in addition to incompleteness, over-specification, etc.). Since those problems are found very early in the life cycle, in particular, before any design, implementation and testing work has been done, this distinguished PBSE feature considerably saves time and money. The SDV phase is in fact entered only when some $\langle \mathrm{Z} \rangle$ that is free from obvious impossibility results has been established.

3.2 The System Design and Validation Phase

The other PBSE phase that occurs before any implementation work on the system is the *System Design and Validation* (SDV) phase, which aims at building the specification [S] of a solution S that provably solves the problem(s) captured in $\langle \mathrm{Z} \rangle$. The outcome of the SDV phase is a *technical leaflet* (TL) for pair $\{\langle \mathrm{Z} \rangle, [\mathrm{S}]\}$, which is a 5-tuple $\{\langle \mathrm{Z}|\mathrm{z} \rangle, [\mathrm{S}|\mathrm{s}], \mathrm{proofs}, \mathrm{Cs}, \mathrm{FD\_Oracle}\}$ consisting of

– the problem specification $\langle \mathrm{Z} \rangle$, with unvalued variables $\langle \mathrm{z} \rangle$,
– the solution specification [S], with unvalued variables [s] (which match $\langle \mathrm{z} \rangle$), usually resting upon some *design assumptions* (DA),
– *proofs* (or pointers to such proofs) that [S] meets $\langle \mathrm{Z} \rangle$,
– *feasibility conditions* (FCs), i.e., analytical conditions that must hold between $\langle \mathrm{z} \rangle$ and [s] in order to ensure that S's valued properties (e.g. response times and availability figures) hold,
– *FD_Oracle*, (the specification of) a computer program that instantiates the FCs in order to simplify and speed-up the FD phase; the FD_Oracle can be developed any time after completion of the SDV phase.

PBSE does not make any requirements on how the properties and models are expressed. Typically, however, Logical Safety properties are expressed as invariants defined over values taken by sets of variables which represent the state of the spacecraft (or more generally, of the CBS). Proofs for Logical Safety or

[3] For the past decade, PBSE/TRDF has been applied without tool support. One of the goals of ASSERT is to develop the prototype of a RC tool (SDV and FD tool prototypes as well), whereby the RC work conducted manually at the beginning of ASSERT would be replayed in a somewhat automated manner.

Liveness are proofs in logic. Timeliness proofs are proofs in Combinatorial Analysis/Scheduling Theory. Dependability proofs are combinations of such proofs, augmented with coverage analysis. In $\langle$p.Z$\rangle$, one finds Cov([S]), which stands for the smallest acceptable coverage to be met by [S], as stipulated by the client.

The Technical leaflet for the whole CBS system or the system family — that is for the pair $\{\langle Z\rangle, [S]\}$ — is inevitably a set of TLs, one for each of its components, which can be either *application-centric building blocks* (ABBs) or *computer-centric building blocks* (CBBs). Typical CBBs deal with system-level issues like distributed resource management, failure detection and/or masking, synchronization, concurrency control, timeliness, etc, whereas ABBs realize the actual functional requirements. CBBs provide the abstraction of a computing environment as "perfect" as required for the ABBs. Quite often, "perfection" means keeping invisible such things as concurrent computations or failures. Application programmers, who design ABBs, can hence concentrate solely on application-related issues, using their favorite formal/informal software engineering methods, and need not worry about specific peculiarities or/and imperfections of the underlying CBS. Moreover, ABBs can be developed and verified in isolation of each other, with no (or quite limited) need for global or integrated verification.

The specification of the solution [S] is in fact a modular specification, which results from building a *design tree* rooted at $\langle$Z$\rangle$: Top-level ABBs are typically just "containers" for application-level functionality. The DAs of such top-level ABBs are hence fairly idealistic, like "there are only perfect processors in the system". Since such assumptions have a rather bad coverage, this leads to corresponding (sub-)problem specification(s) to be met by the specifications of novel or reused CBBs, which must be dealt with at the next (lower) level of the design tree. This process of successive refinement proceeds along some number of branches. On a given branch, one stops designing whenever both (1) the specification arrived at is deemed implementable and (2) its DAs have a coverage at least as high as Cov([S]).

Another important output of the SDV phase are the FCs, which are typically a set of constraints required for the solution [S] to work. In order to simplify checking of the feasibility of some particular dimensioning of the problem variables $\langle$z$\rangle$ and calculating the corresponding dimensioning of [s], FCs are instantiated as a computer program (referred to as an *FD_Oracle*, valid for pair $\{\langle Z\rangle, [S]\}$), which can be developed any time after completion of the SDV phase.

3.3 The Feasibility and Dimensioning Phase

For any given problem $\langle$Z$\rangle$, the SDV phase leading to [S] — as well as *implementation & unitary verification*, which is not a PBSE activity — is conducted only once, i.e., [S] for $\langle$Z$\rangle$ needs to be established and proved only once.

By contrast, the *Feasibility and Dimensioning* (FD) phase (as well as the following *instantiation* phase and the IT phase) has to be conducted every time a specifically customized release of [S] is to be fielded. The FD phase consists in a user choosing some specific valuation $Val(\langle z\rangle)$ of the unvalued problem variables in $\langle$Z$\rangle$, running the FD_Oracle, and (if possible) obtaining the resulting valuation $Val([s])$ of the unvalued system/solution variables in [S]. For example, this is how one knows the smallest period of activation of a scheduler, the smallest memory space for a waiting queue, or the smallest degree of redundancy which are necessary for meeting the required reliability figures. If the FCs are violated, the FD_Oracle indicates the reasons why, in which case some of the values assigned to $\langle z \rangle$ must be "relaxed" (e.g., some deadlines augmented).

3.4 The Integration Testing Phase

As stated above, *implementation & unitary verification* is not a PBSE activity. However, in order to maintain a continuous chain of proofs, not only from $\langle Z \rangle$ to [S], but also from [S] to S, automatic code generation and formal (local) verification should be used also during implementation of S.

When implementation and unitary verification has been conducted for all BBs that are part of S, the *Integration Testing* (IT) phase can be performed in order to check whether the composition of BBs is correct — a daunting task under current practice, since exponential complexity is to be faced. This is not the case under PBSE, which eliminates the classic state explosion problem involved in global verification and improves the achieved coverage compared to current testing practices, respectively, for two reasons essentially:

– No or just some limited global verification is necessary (proofs replacing possibly huge sets of tests).
– The suite of tests to be performed can be generated (if so desired) as a by-product of running the FD_Oracle, rather than by "guessing" them.

Consequently, with PBSE, integration testing work is typically unnecessary or at least limited, since "composition correctness" has been proved during the SDV phase (otherwise, [S] would not exist).

4 A Design and Reuse Example

Among the major advantages of PBSE, which is also a major target of ASSERT, is its potential for re-using BBs specified and designed in former projects. Given that (1) PBSE is concerned about system-level problems, which appear over and over again in many different applications, and (2) reuse in PBSE also includes reusing the design specifications and the proofs, the potential for reuse is indeed high.[4] Thanks to the TLs, the common practice of developing everything from scratch and/or best-effort reuse of existing components can be replaced by a systematic reuse exercise according to PBSE principles, i.e. conditioned upon using provably correct compositions of components.

A major goal of ASSERT in this realm is the definition of *system families* (SF), which represent a reasonably large class of space applications that share a

[4] Consequently, the budget and time savings that can be achieved with PBSE are high as well.

sufficiently large set of common properties. Ideally, when a new application is to be developed from a system family, most of the family's generic BBs are reused, and only the few ones that encapsulate application-specific functionality need to be modified/added.

One of the pilot projects in ASSERT is devoted to the definition of a system family for satellite missions. It is based upon a generic specification $\langle \text{Z(SF)} \rangle$ that captures the CBS problem common to such missions. ASSERT shall end up with the design specification (+ implementation) of a solution [S(SF)], consisting of a set of generic BBs, that matches $\langle \text{Z(SF)} \rangle$. In order to develop a particular satellite mission, say, a telecommunications satellite (TS), which is known to belong to SF (since $\langle \text{Z(TS)} \rangle \equiv \langle \text{Z(SF)} \rangle$), a user simply decides on some valuation $Val(\langle \text{z} \rangle)$ mirroring the TS-centric instantiation of SF and runs the FD_Oracle that was built for $\langle \text{Z(SF)} \rangle$, [S(SF)]. In other words the user has to conduct the FD and IT phases only. If $\langle \text{Z(TS)} \rangle \not\equiv \langle \text{Z(SF)} \rangle$, then some SDV work is necessary, reusing the BBs developed for SF. Consequently, the design tree for TS quickly reaches nodes that are already available. As a consequence, real design and implementation work is only needed for features that are specific for TS.

More generally, assume that, at some node of the SDV tree rooted at $\langle \text{Z(TS)} \rangle$, one is contemplating the specification of a sub-problem $\{\langle \text{m.Z(X)} \rangle, \langle \text{p.Z(X)} \rangle\}$, and that there is an existing TL matching this specification (searches for matching TLs will be done by an SDV tool in the future). Since the specification [S] found in this TL has been proved correct for $\langle \text{Z(X)} \rangle$, the corresponding solution can simply be reused as such (with or without prior dimensioning), provided the conditions for stopping the SDV work for that SDV tree node are met.

For example, consider that $\langle \text{Z(X)} \rangle$ is the specification of some problem that was addressed in the A3M project[5]. The A3M objective was to develop a new generation of generic components as basic building blocks for the development of middleware targeting various on-board space applications [20]. The core CBBs developed in A3M employ asynchronous distributed fault-tolerant algorithms [21] for distributed consensus, coordination, and atomic commit, which are built atop of Chandra/Toueg unreliable failure detectors [22]. They rest upon design assumptions such as processor crashes, arbitrarily variable delays, and reliable communications, but do not need any notion of global time in the system. Thus the logical safety and liveness properties stated in $\{\langle \text{m.Z(X)} \rangle, \langle \text{p.Z(X)} \rangle\}$ hold with these CBBs regardless of the (implementation-dependent) timing properties of the underlying system.

When the design assumptions meet the conditions for stopping the SDV work, then A3M solution can be reused in ASSERT as such. If some design assumption, say, Y, does not meet the conditions for stopping the SDV work (e.g., if processor omission failures and/or unreliable communications are assumed for the ASSERT SF), then Y translates into a sub-problem $\{\langle \text{m.Z(Y)} \rangle, \langle \text{p.Z(Y)} \rangle\}$ (e.g., simulating processor crashes in the presence of omissions and providing reliable communications over unreliable channels), and the SDV work continues.

[5] Advanced Avionics Architecture and Modules, conducted by EADS Astrium, INRIA, LAAS, Axlog Ingenierie and funded by ESA/ESTEC (2001–2003).

5 The Virtual System Model

The PBSE requirements capture phase poses some particular challenges in that it targets inherent application needs only, rather than (premature) design considerations. In fact, freezing requirements actually rooted in traditional or even anticipated solutions in ⟨Z⟩ unnecessarily restricts the solution space for the later SDV work. This problem became particularly apparent during the RC phase for the complex system families/pilot projects in ASSERT, which was conducted by multidisciplinary teams: Following industrial practice, and quite natural for engineering disciplines, the initial versions of the URDs were heavily populated with a priori chosen system architectures, failure management strategies, process synchrony assumptions and other design considerations. Extracting out exactly those requirements that must be fulfilled by a CBS in order to meet the demands of the particular application (but nothing else) turned out to be a challenging task, cf. [13].

5.1 Using the VS Model for RC

In order to alleviate this problem, we introduced the *virtual system model* (VS model) for requirements capture. The virtual system model consists of several levels, which represent different levels of abstraction of a computer-based system. A level is populated by entities that represent components of a CBS at the corresponding level of abstraction, i.e., can be seen as a suitable "projection" of a CBS onto some specific abstraction level. Consequently, the VS model can be employed for reasoning about a yet-to-be-designed system as well.

The levels foreseen in the VS model may be domain-dependent. In ASSERT, the following levels have been identified to be necessary and sufficient for embedded systems in the aerospace domain: *Equipment & Humans level* (EH-Level), *Application level* (AP-Level), *Middleware level* (MW-Level), *Basic Service level* (BS-Level), and *Hardware level* (HW-Level).

The EH-Level provides the highest level of abstraction. It "connects" a CBS with its environment. It is populated with entities that may or may not be considered part of the CBS. They include external equipment, sensors and actuators as well as human users. Although in the implemented system EH-Level entities are connected by means of the HW-Level, which encompasses the raw computing and communication hardware of a CBS, in the VS model the EH entities can directly interact with entities at any level.

The AP-Level is made up of all the entities that instantiate the application's semantics. They are distributed/partitioned according to functional analysis considerations.

MW-Level entities typically serve two purposes: First, they provide a level of encapsulation, which allows application-level entities to access resources in a way that is independent of their physical location. Second, they typically host all the distributed fault-tolerant algorithms and protocols needed to "solve" ⟨Z⟩, i.e., the system-level algorithms and protocols specified within [S].

The BS-Level is populated with entities that augment/encapsulate the raw services provided by HW-Level entities in a generic manner, in order to provide universal and elementary services needed by entities residing at higher levels. Typically, BS entities are operating systems, real-time kernels, link communication protocols, TCP-like communication protocols, I/O handlers, memory access/management protocols, etc.

Finally, the HW-Level is populated with entities which provide the physical capability to execute programs (SW, firmware, gate-level compiled code, etc.), and to exchange bits over physical communication entities (both on-board and long-haul communications, communications with sensors and actuators, etc.). In other words the HW-Level provides the raw "execution machinery", but does not include programs written in HW, such as the logic in gate level compiled code of an ASIC.

In fact, although the VS levels above appear to follow traditional implementation levels, it is important to understand that they are not meant to imply any particular implementation, since an AP-Level entity may actually be implemented as a real AP-Level SW process, or as a triple {AP-Level SW component, BS-Level SW component, HW-Level HW component}, and might even involve on-line reconfiguration. Moreover, VS levels do not have any particular hierarchical relationship. They must rather be viewed as sets of orthogonal entities that may have all kinds of mutual interactions: AP-Level entities in the VS model are not restricted to interact solely with the MW-Level in order to invoke services, nor do MW-Level entities provide services to the AP-Level only. For example, a BS-Level entity — and even a HW-Level entity — may invoke a service at the MW-Level.

Consequently, the VS model used for conducting a PBSE RC phase is generic, in the sense that it does not carry any restrictions relative to the construction of ⟨Z⟩ or future SDV work leading to [S]. Hence, it can indeed be employed for capturing ⟨Z⟩ for a yet-to-be-designed system in the PBSE RC phase.

The VS model opens up another level of "separation of concerns", beyond PBSE's ability to deal with an application's functional and non-functional aspects independently of each other: It allows to capture models and properties at every VS level independently and in strict isolation of each other. This leads to a significant reduction of the overall complexity of the PBSE RC phase and allows even further parallelization of the RC work, which reflects reality as experts in AP-Level software are most likely not experts in space-compliant hardware.

More specifically, the whole set of application requirements can be mapped onto (or rather: "sliced" according to) the different levels of abstraction corresponding to the VS levels. For every level, the resulting projections ("slices") can then be captured independently of the other levels. Note that properties are always associated with the level where they are required to hold. If, for instance, some AP level processes shall enjoy the ACID properties of transactions, the atomicity, concurrency, isolation, and durability properties [19] are captured for the AP-Level in ⟨p.Z⟩. Although those properties are likely to be provided by MW-Level concurrency control algorithms in the yet-to-be-developed solution

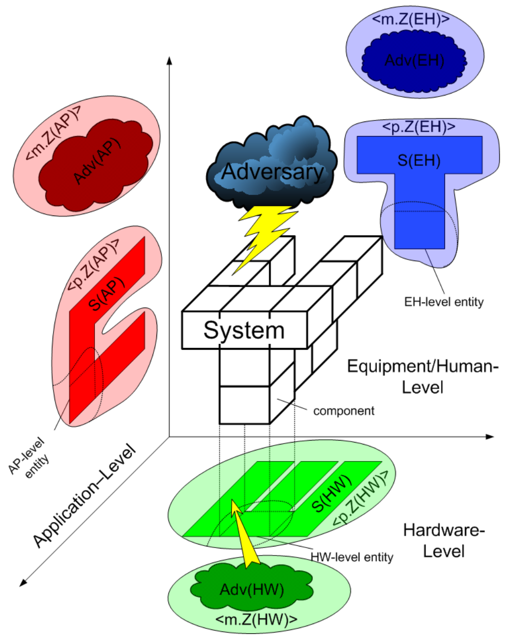

Fig. 2. *Visualization of the relation between system requirements and designed system projected onto VS levels: The designed system must provide properties that are fully within the specified properties when exposed to an adversary that is not stronger than the specified models*

[S], given the currently affordable technology, this shall not be frozen at RC time. Hence, the ACID properties, which are captured at the AP-Level, are not captured *for* the MW-Level. Any entity E of any level that has a TL showing that E provides the ACID properties is a correct BB for providing those properties at the AP-Level.

5.2 Using the VS Model for SDV

Figure 2 visualizes the resulting orthogonalization of the RC capture phase enabled by the VS model. Consider the AP-Level, for example. Let ⟨m.Z(AP)⟩ and ⟨p.Z(AP)⟩ denote the models and properties captured for this level, i.e., the projections of the whole set of requirements "hidden" in the application's URD onto the AP-Level. PBSE SDV work must eventually ensure that, whenever the HW-Level faces an adversary Adv(HW) that is not stronger than specified in ⟨m.Z(HW)⟩, the behavior of the forthcoming solution [S] (and hence the fielded system S) projected onto the AP-Level must stay within the behaviors stated in ⟨p.Z(AP)⟩.

Of course, the SDV work that provides the solution specification [S] must eventually consider all combinations of models captured in ⟨m.Z⟩ in order to consider the worst-case failure occurrences at all levels *simultaneously*, for example.

That is, in sharp contrast to the RC phase, where everything can be considered in isolation, the SDV phase must deal with the combinatorial complexity of combining the independently captured models and properties, including combined failure occurrences, worst-case event arrivals, etc.

From the virtual system model point of view in ASSERT, the most important levels for RC capture are the AP- and EH-Level, since entities at these levels are in fact the "end users" of the CBS. Note that, for the EH-Level, only the specification (models and properties) with respect to the interface(s) with the CBS is of concern.

If there was no reuse of pre-existing products, nothing would have to be captured at the levels below the AP-Level. Since systems are almost never implemented from scratch, however, the RC phase is also concerned with the identification of models and properties that characterize pre-existing BBs at any level, especially at the HW-Level and BS-Level. Any pre-existing product is either *trusted* or *not trusted*. By definition, at the time of writing, all existing products have been developed in some former projects *without* applying PBSE. Trusted products are those which have been developed and tested following certain rules, typically those stipulated by agencies or enforced by certification bodies (e.g., DO-178 or IEC standards, SILs in the UK).

Of course, although such products are considered trusted on the basis of careful design, diversified redundancy, sufficient testing, space-compliance and other means, this does not imply that they are fault-free. Nevertheless, the nature of the technical data available for a trusted product makes it possible to do some reverse PBSE work, i.e., to construct its TL a posteriori, at least partially (its models and properties, as well as its design assumptions). This way, pre-existing trusted products can be incorporated in ⟨Z⟩ and [S].

6 Concluding Discussion of PBSE

To achieve its ambitious goal of facilitating provably correct and reusable engineering work for critical fault-tolerant distributed real-time embedded systems, PBSE combines a number of different features in a common framework. First of all, a dedicated requirements capture phase has to be conducted, which provides an agreed-upon specification of the problem ⟨Z⟩ to be solved. ⟨Z⟩ not only describes what is to be achieved (properties ⟨p.Z⟩), but also under which conditions/circumstances (models ⟨m.Z⟩). The VS model has been introduced as an effective means to capture those properties and models in isolation of each other.

In general, conducting a requirements capture phase is difficult, for several reasons, such as: The intertwining of requirements and solutions, conflicting requirements of different stakeholders or the need for early freezing of requirements (waterfall model). PBSE does not suffer from those problems, however, for two reasons essentially: First, PBSE focuses solely on non-functional CBS-centric issues, which are reasonably independent of the particular application requirements. Second, the unvalued problem and solution variables ⟨z⟩ and [s] allow to introduce a user-decided degree of genericity in ⟨Z⟩ and [S], respectively, which effectively eliminates the well known problem having to nullify SDV work whenever $\langle Z \rangle$ is changed.

Finally, rather than on "a posteriori" global verification, PBSE rests upon "a priori" proof obligations to be fulfilled in the course of system design activities. PBSE hence necessarily saves time and money due to the fact that it provides solutions that are correct-by-construction; no time and money are wasted on trial and error-detection-and-correction iterations. Additionally, the concept of reuse also goes beyond what is commonly associated with this term: Not only existing implementations of BBs can be reused, but also their designs and proofs. Component reuse in PBSE is in fact similar in spirit to the use of existing lemmas for proving a new theorem in mathematics.

There is the widespread belief that proofs are too difficult to do in daily practice, a problem that has also slowed down the acceptance of formal software engineering methods. However, system engineers are not supposed to "do the proofs" (unless they run into a non-generic or unknown system problem). A fully developed PBSE process will eventually allow engineers and technicians to simply use instruction manuals, supported by appropriate tools, as is the case in other fields with good and mature engineering practice [23]: In handbooks for electricians, for example, one finds rules for how to install derivations, for computing voltages, etc. It is never the case that an electrician is asked to demonstrate anew the correctness of his doings based on Ohm laws or Kirchoff laws. Nevertheless, rules of good practice in mature engineering domains rest entirely upon such scientific results. We anticipate that this is going to happen to system engineering for computer-based systems as well: Rather than being founded on "experience, "intuition", or "good sense", rules of good system engineering practice will eventually rest upon science, and PBSE has been developed for reaching this goal.

References

1. Le Lann, G.: Proof-based system engineering and embedded systems. In: European School on Embedded Systems (Veldhoven, NL, Nov. 1996). Volume 1494 of Lecture Notes in Computer Science., Springer-Verlag Pub. (1998) 208–248 invited paper.
2. Le Lann, G.: Models, proofs and the engineering of computer-based systems: A reality check. In: Proc. 9th Annual Intl. INCOSE Symposium on Systems Engineering: Sharing the Future. Volume 4. (1999) 495–502 Best Paper Award.
3. Chung, L., Nixon, B.A., Yu, E., Mylopoulos, J.: Non-Functional Requirements in Software Engineering. Kluwer Academic Publishing (2000)
4. Kuhn, D.: Sources of the failure in the public switched telephone network. IEEE Computer **30** (1997) 31–36
5. Inquiry Board Report: ARIANE 5 — Flight 501 Failure. (1996) Available online `http://ravel.esrin.esa.it/docs/esa-x-1819eng.pdf`.
6. Le Lann, G.: An analysis of the ariane 5 flight 501 failure—a system engineering perspective. In: Proceedings of the IEEE International Conference and Workshop on Engineering of Computer-Based Systems. (1997) 339–346
7. Le Lann, G.: The failure of satellite launcher ariane 4.5. Safety Critical Mailing List at `http://www.cs.york.ac.uk/hise/text/sclist/lelannariane.html` Archived Contributions, Contribution on the Failure of Ariane 5 flight 501 (1999)

8. Zave, P., Jackson, M.: Four dark corners of requirements engineering. ACM Trans. Softw. Eng. Methodol. **6** (1997) 1–30
9. Jackson, M.: Software requirements & specifications: A lexicon of practice, principles and prejudices. ACM Press/Addison-Wesley Publishing Co. (1995)
10. Zave, P.: Classification of research efforts in requirements engineering. ACM Comput. Surv. **29** (1997) 315–321
11. Nuseibeh, B., Easterbrook, S.: Requirements engineering: A roadmap. In: ICSE '00: Proceedings of the Conference on The Future of Software Engineering, ACM Press (2000) 35–46
12. Jackson, M.: The meaning of requirements. Ann. Software Eng. **3** (1997) 5–21
13. Swartout, W., Balzer, R.: On the inevitable intertwining of specification and implementation. Commun. ACM **25** (1982) 438–440
14. Sabetzadeh, M., Easterbrook, S.: Analysis of inconsistency in graph-based viewpoints: A category-theoretic approach. In: Proceedings of the 18th IEEE International Conference on Automated Software Engineering. (2003) 12–21
15. Stevens, R., Brook, P., Jackson, K., Arnold, S.: Systems Engineering: Coping with complexity. Prentice Hall (1998)
16. Meyer, B.: Applying design by contract. IEEE Computer **25** (1992) 40–51
17. Abrial, J.: The B Book. Cambridge University Press (1996)
18. Lynch, N.: Distributed Algorithms. Morgan Kaufman (1996)
19. Bernstein, P., Hadzilacos, V., Goodman, N.: Concurrency Control and Recovery in Database Systems. Addison-Wesley Publishing Company (1987)
20. Honvault, C., Le Roy, M., Gula, P., Fabre, J.C., Le Lann, G., Bornschlegl, E.: Novel generic middleware building blocks for dependable modular avionics systems. In: Proceedings of the 5th European Dependable Computing Conference (EDCC-5). Volume 3463 of LNCS., Budapest, Hungary, Springer Verlag (2005) 140–153
21. Hermant, J.F., Le Lann, G.: Fast asynchronous uniform consensus in real-time distributed systems. IEEE Transactions on Computers **51** (2002) 931–944
22. Chandra, T.D., Toueg, S.: Unreliable failure detectors for reliable distributed systems. Journal of the ACM **43** (1996) 225–267
23. Maibaum, T.: Mathematical foundations of software engineering: A roadmap. In: ICSE '00: Proceedings of the Conference on The Future of Software Engineering, ACM Press (2000) 161–172