Requirements Eng (2003) 8: 222–235
DOI 10.1007/s00766-002-0159-4

ORIGINAL ARTICLE

K. Suzanne Barber · Tom Graser · Jim Holt
Geoff Baker

Arcade: early dynamic property evaluation of requirements using partitioned software architecture models

Received: 22 May 2002 / Accepted: 15 September 2002 / Published online: 3 June 2003
© Springer-Verlag London Limited 2003

Abstract A fundamental goal of software engineering research is to develop evaluation techniques that enable analysis early in the software development process, when correcting errors is less costly. The Systems Engineering Process Activities (SEPA) Arcade tool employs a number of techniques to evaluate dynamic properties of requirements including correctness, performance, and reliability. To mitigate a number of practical issues associated with dynamic property evaluation, Arcade leverages the SEPA 3D Architecture, a formal requirements representation that partitions requirements types amongst a set of interrelated architecture models. This paper presents a case study illustrating how Arcade uses the SEPA 3D Architecture to help manage complexity associated with dynamic property evaluation, to reduce the level of evaluation technique expertise required to perform dynamic property evaluations, and to support an iterative, incremental approach that allows early evaluation using partial requirements models.

Keywords Dynamic properties · Requirements models · Software architecture

1 Introduction

Software architecture models are representations of software requirements intended to prescribe a blueprint for system implementation [1, 2]. Unlike natural-language requirements representations, model-based requirements representations enable a variety of techniques for evaluating requirements early in the software lifecycle, when detection and correction of errors are less costly [3, 4]. Early evaluation results also provide valuable insights that can aid in managing requirements evolution, and can serve as guidance for system implementation.

To encourage software architectures as a viable framework for modelling and evaluating requirements, researchers in the Laboratory for Intelligent Processes and Systems at the University of Texas at Austin (UT:LIPS) have developed the Systems Engineering Process Activities (SEPA) methodology. The SEPA methodology partitions requirements types among a set of interrelated software architecture models collectively referred to as the SEPA 3D Architecture: Domain Reference Architecture (DRA), Application Architecture (AA), and Implementation Architecture (IA). The DRA specifies *domain requirements* (e.g., business process functionality, data and their relationships, timing between functions), the AA specifies *non-functional requirements* (e.g., application look-and-feel, runtime performance requirements), and the IA specifies *installation requirements* (e.g., available site hardware platforms, middleware, and communications software). The SEPA methodology has been successfully employed on large distributed development projects [5], and is currently being applied to an e-business project within Motorola called *eDesign*.

This paper presents Arcade. Arcade enables early dynamic property evaluations of requirements specified in software architecture models. Dynamic properties are properties that are manifested during system execution, including *correctness*, *performance*, and *reliability*. Early evaluations of software architecture models cannot provide *quantitative* measures of ultimate system properties, since many factors encountered later in the software lifecycle affect properties of the system, for example choice of algorithms, technologies, or computing platform [1]. Nevertheless, early evaluation can provide a *qualitative* view of properties that are governed by the requirements specified in software architecture models. Previous publications have covered in detail the techniques employed by Arcade for dynamic

K. S. Barber (✉) · T. Graser
Laboratory for Intelligent Processes and Systems, University of Texas at Austin, Austin, Texas, 78712-1084, USA
E-mail: barber@lips.utexas.edu

J. Holt · G. Baker
Motorola, 7700 West Parmer Lane, Austin, Texas, 78729, USA

property evaluations [6, 7, 8]. The main contribution of this paper is to illustrate the benefits of the Arcade approach in the context of an industrial case study. This case study illustrates that Arcade's qualitative evaluation results can serve several valuable purposes, including (1) aiding in detection and correction of requirements errors, (2) aiding in requirements evolution, and (3) providing guidance to system implementers.

To provide a systematic, automated approach for early dynamic property evaluation of requirements, Arcade leverages the SEPA 3D Architecture in conjunction with a number of dynamic property evaluation techniques, including model checking, discrete event simulation, and probabilistic graph model algorithms. In doing so, Arcade addresses several practicality issues associated with the selected evaluation techniques. These practicality issues can be divided into *expertise-related* issues and *capacity-related* issues.

Expertise-related issues addressed by Arcade are: (1) automated translation of requirements specified in a software architecture model into a format suitable for dynamic property evaluation techniques, and (2) automated collection and presentation of evaluation results to stakeholders in intuitive formats. Thus, Arcade enables stakeholders to perform dynamic property evaluations, to understand the evaluation results, and to make decisions based upon those results.

Arcade addresses capacity-related issues by leveraging partitioned software architecture models (e.g., the SEPA 3D Architecture) to reduce the complexity of evaluation. This is important because complexity affects: (1) the human capacity to produce and work with large models and associated evaluation results, and (2) computer capacity issues such as CPU and memory requirements [9, 10]. Arcade's support for partitioned software architecture models mitigates complexity by allowing dynamic property evaluation to be performed early (e.g., using partial requirements specifications), and iteratively (e.g., using an incremental approach to requirements modelling and evaluation).

The remainder of this paper is organised as follows. Section 2 describes the SEPA 3D Architecture and how Arcade leverages its partitioned software architecture models to provide early dynamic property evaluations. The benefits of performing early dynamic property evaluations are then illustrated in Sect. 3, using the eDesign DRA as a case study. Section 4 discusses related work, and Sect. 5 presents conclusions.

2 The SEPA 3D Architecture

The SEPA 3D Architecture provides a framework for representing and evaluating different types of requirements from multiple stakeholders, including domain experts, end users, application developers, and system integrators. Section 2.1 discusses how the formal, computational representation of requirements embodied by the SEPA 3D Architecture facilitates requirements evolution and reuse. Section 2.2 describes how Arcade leverages the SEPA 3D Architecture to enable dynamic property evaluation and to mitigate practicality issues associated with dynamic property evaluation (described in Sect. 1). Section 2.3 presents the Domain Reference Architecture (DRA) metamodel, and Sect. 2.4 introduces the eDesign DRA.

2.1 Requirements evolution and reuse

While requirements analysis and software architecture modelling typically precede design and implementation in the software lifecycle, requirements evolution is a reality, both during development and following deployment [11]. Furthermore, observations from a previous empirical study conducted by the authors indicate that different types of requirements (e.g., domain functionality, application look-and-feel, installation constraints) evolve at different rates [12]. For example, domain requirements (functional, data, timing requirements) may remain stable over the course of multiple technology cycles. This stability within requirements types provides reuse opportunities. For example, stable domain requirements can be reused as non-functional requirements associated with specific technologies or installation sites evolve ("sites" are characterised as specific type of users as well as the infrastructure environment where the system will reside).

Recognising that requirements types evolve at different rates and that reuse opportunities align with requirements type boundaries, the SEPA methodology delivers a comprehensive approach to partition types of requirements into a set of related software architecture models [13]. Derivation of the respective SEPA architectures is driven by these requirements types (Fig. 1). The Domain Reference Architecture (DRA) is derived

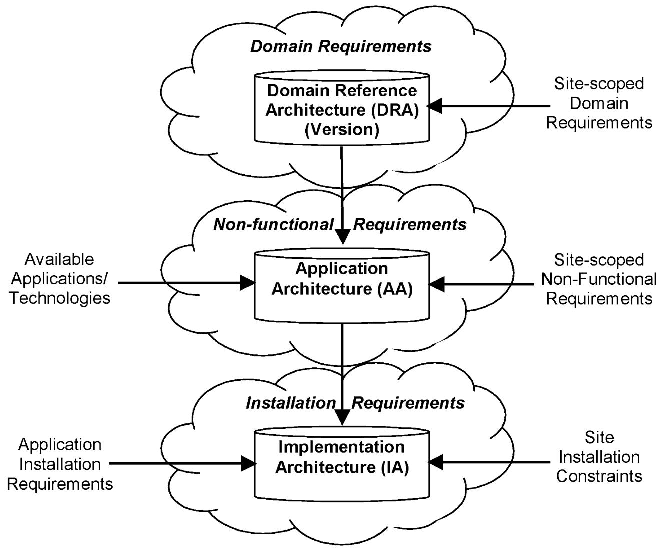

Domain Requirements
Domain Reference Architecture (DRA) (Version)
Site-scoped Domain Requirements
Non-functional Requirements
Available Applications/ Technologies
Application Architecture (AA)
Site-scoped Non-Functional Requirements
Installation Requirements
Application Installation Requirements
Implementation Architecture (IA)
Site Installation Constraints

Fig. 1 DRA, AA, and IA relationships

from domain functional, data, and timing requirements. For a specific customer site or type of customer site these requirements are scoped, yielding a DRA Version customised for that site. The Application Architecture (AA) is created by correlating application technology components to the domain requirements in the DRA Version and the non-functional requirements identified by the customer. The Implementation Architecture (IA) represents a refinement of the AA through the addition of site installation requirements. The IA also considers the installation requirements and compatibility of applications selected for the AA.

2.2 Leveraging model partitioning

The ability to partition requirements types amongst the DRA, the AA, and the IA allows for early, iterative dynamic property evaluation. For example, because the DRA is a highly abstract representation of functionality that is independent of any particular implementation, DRA evaluation can uncover errors associated with domain requirements early in the requirements modelling and analysis process. Subsequently, the AA and IA can be used to evaluate design tradeoffs and site configuration implications once customer site requirements are known.

Arcade, in conjunction with the SEPA 3D Architecture, supports partitioning in two dimensions. The first partitioning dimension is defined by the SEPA 3D Architecture: partitioning *across* requirements types (e.g., the DRA, AA, and IA models). For example, evaluating only the requirements specified in a DRA reduces the complexity of the behavioural specification by expressing system behaviour at an abstract domain level (e.g., business process functionality, data and their relationships, timing between functions), while deferring evaluation of requirements specified in the AA and IA. The second partitioning dimension is defined by partitioning *within* a single requirements type to create a *subset model*. For example, a subset model of a DRA might represent only the domain requirements necessary to accomplish a single domain process, thereby focusing evaluation on specific functionality of interest early during the requirements acquisition and modelling process, before requirements are completely modelled. In addition to mitigating complexity, the incremental approach supported by these partitioning dimensions can enable the ability to evaluate the impact of requirements changes (i.e., requirements evolution), and can be used to highlight specific requirements dependencies.

Using both partitioning dimensions with the eDesign requirements (e.g., evaluating eDesign DRA requirements in isolation from the AA and IA requirements, and evaluating eDesign DRA subset models) presented opportunities to assess correctness of domain requirements and to discover valuable information about the dynamic property characteristics of the domain (see case study in Sect. 3). While it is not presented in this paper, subsequent evaluation of the eDesign AA and IA yielded additional information concerning the effects of non-functional requirements and installation requirements on the dynamic properties of the system.

2.3 The SEPA Domain Reference Architecture

The SEPA Domain Reference Architecture (DRA) is composed of Domain Reference Architecture Classes (DRACs), each of which specifies some portion of domain data and functionality. These classes and their relationships become a reusable blueprint that guides development efforts in terms of (1) the functional, data, and timing (i.e., ordering of functions) requirements to be satisfied, and (2) prescribed architectural structure specifying collections of and dependencies between (i.e., data or timing) system functionality. Each time the blueprint is reused for a new system development effort, DRACs may be instantiated by different applications (i.e., implementation solutions).

The functionality and data allocated to a DRAC, and the interrelationships between DRACs, are represented using a meta-model composed of the three submodels shown in Fig. 2: the Declarative Model (D-M) defines the attributes (i.e., data and events) and services (i.e., functionality) that should be offered by an instance of the DRAC specification; the Behavioural Model (B-M) describes the behaviour expected from an instance of the DRAC through a high-level state chart; and the Integration Model (I-M) defines the constraints and dependencies between DRAC instances resulting from the distribution of dependent domain functions across DRACs. These dependencies are an artifact of the input and output of data and events among DRAC services (i.e., domain Function1 receives EventX from domain Function2) and are described in predicates capturing service pre- and post-conditions.

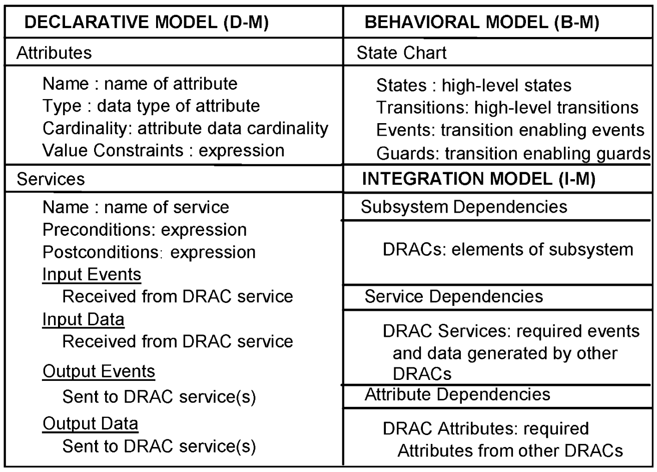

DECLARATIVE MODEL (D-M)
BEHAVIORAL MODEL (B-M)
Attributes
State Chart
Name : name of attribute
Type : data type of attribute
Cardinality: attribute data cardinality
Value Constraints : expression
States : high-level states
Transitions: high-level transitions
Events: transition enabling events
Guards: transition enabling guards
Services
INTEGRATION MODEL (I-M)
Name : name of service
Preconditions: expression
Postconditions: expression
Input Events
Received from DRAC service
Input Data
Received from DRAC service
Output Events
Sent to DRAC service(s)
Output Data
Sent to DRAC service(s)
Subsystem Dependencies
DRACs: elements of subsystem
Service Dependencies
DRAC Services: required events and data generated by other DRACs
Attribute Dependencies
DRAC Attributes: required Attributes from other DRACs

Fig. 2 Domain Reference Architecture class meta-model

2.4 Case study domain: the eDesign system

The eDesign system is being developed by Motorola's Semiconductor Products Sector (SPS) to efficiently deliver SPS product technical information and collateral products to internal and external Motorola customers. Major functionality in the eDesign system includes Document Authoring, Document Configuration Management, Content Administration, and Content Delivery (Fig. 3). eDesign is being deployed with a mix of off-the-shelf as well as in-house developed applications. The SEPA methodology and its supporting tools were chosen because the eDesign project is being developed in a very iterative, rapid fashion that could benefit from SEPA's approach to requirements evolution. Furthermore, in the e-business domain the business needs (e.g., domain requirements) are evolving at a less rapid pace than technology requirements (e.g., the AA and IA). Therefore, the goals of this effort were: (1) to provide the eDesign team with a solid requirements foundation upon which to rapidly evolve their system, (2) to characterise the dynamic properties governed by requirements so that the eDesign team can make better decisions related to evolving technology requirements, and (3) to provide system implementation guidance.

The first step in eliciting the domain requirements of the eDesign system was to perform Requirements Acquisition (RA) to acquire domain usage profiles describing functionality in the domain (two of these usage profiles are used as examples in Sect. 3, and are depicted in Table 1). Usage profiles help to explicitly define and consequently scope the domain functionality, data, timing, interactions, and user types. A usage profile is composed of one or more domain tasks, where a task specification includes the name of a performer capable of executing the task, input data/events required for execution, output data/events produced, and pre-/post-conditions defining necessary conditions to begin execution and expected conditions following execution, respectively.

The full set of usage profiles acquired during RA sessions provided the basis from which an initial DRA version was derived (Fig. 3). Subsequent requirements were gathered to support the definition of the eDesign AA and IA models. Following construction of the initial eDesign DRA, the team proceeded to evaluate its dynamic properties using Arcade. Section 3 presents this work.

3 Evaluating dynamic properties with Arcade

Arcade supports evaluation of three categories of dynamic properties: *correctness*, *performance*, and *reliability*. Because no single evaluation technique is suitable for evaluating all three categories of dynamic properties, Arcade employs a number of techniques, including model checking, discrete event simulation, and probabilistic graph models [6, 7, 8]. The choice of techniques was based upon two criteria: (1) suitability of the technique for evaluating the dynamic properties of interest, and (2) availability of tools implementing evaluation techniques. For the Arcade research it was sufficient to choose well-established techniques that could demonstrate the efficacy of early evaluation using partitioned software architecture models. The selection of a particular set of techniques for the Arcade research does not limit similar approaches from being employed with other evaluation techniques (for example, process algebras, queuing network models).

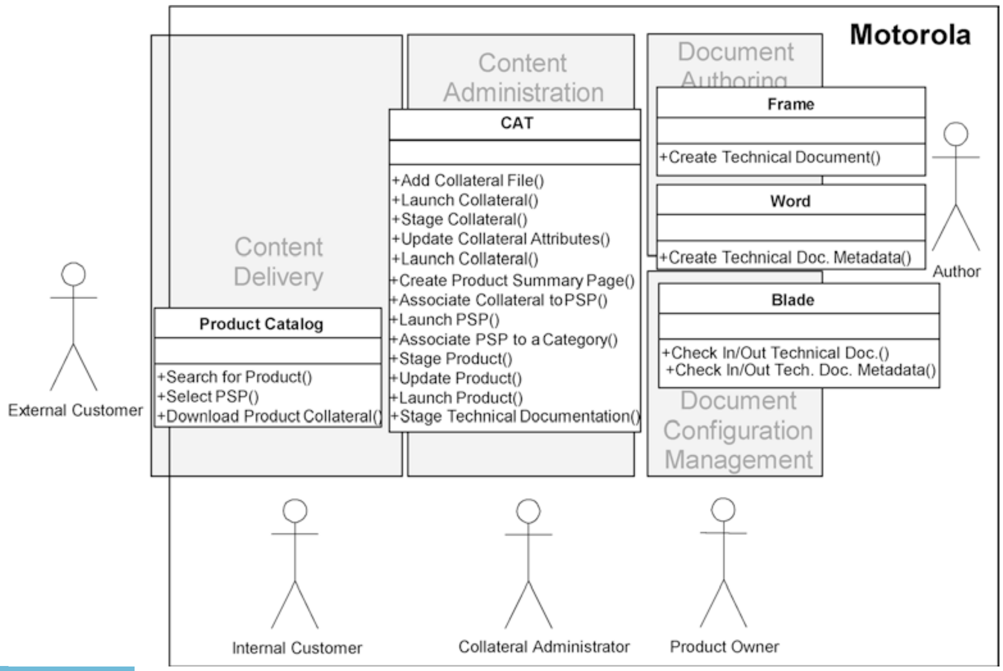


Fig. 3 eDesign domain and stakeholders

Table 1 eDesign usage profiles

Usage profile:	Publish new technical document	**Usage profile**:	Access product information
Task:	Create technical documentation **Performer**: author **Input data/events**: product specification **Output data/events**: technical documentation **Pre-condition**: request for technical documentation **Post-condition**: technical document created	**Task**:	Search for product **Performer**: customer **Input data/events**: product name or taxonomy search criteria **Output data/events**: product list **Pre-condition**: customer requires product info and product state is "launched" and product is assigned to a search category **Post-condition**: product list selected or request cancelled
Task:	Create technical document metadata **Performer**: author **Input data/events**: document attributes, parametric data **Output data/events**: technical document metadata **Pre-condition**: technical document created **Post-condition**: technical document metadata created	**Task**:	Select product summary page (PSP) **Performer**: customer **Input data/events**: product list **Output data/events**: PSP **Pre-condition**: product list selected **Post-condition**: PSP displayed or request cancelled
Task:	Check in technical documentation **Performer**: author **Input data/events**: technical documentation **Output data/events**: none **Pre-condition**: technical document created **Post-condition:** technical document under configuration management	**Task**:	Download product collateral **Performer**: customer **Input data/events**: PSP **Output data/events**: product collateral Information **Pre-condition**: PSP displayed and collateral state is "launched" **Post-condition**: product collateral downloaded or request cancelled
Task:	Check in technical document metadata **Performer**: author **Input data/events**: technical document metadata **Output data/events**: none **Pre-condition**: technical document metadata created **Post-condition**: technical document metadata checked in		
Task:	Change product state **Performer**: publisher **Input data/events**: technical document, technical document metadata **Output data/events**: none **Pre-condition**: technical document checked in and technical document metadata checked in **Post-condition**: technical document state is "staged"		

To assist non-experts in executing dynamic property evaluations and conducting early analysis, Arcade automates (1) translating requirements specified in a software architecture model into a format suitable for use by evaluation techniques, and (2) collecting and formatting evaluation results into intuitive presentations on behalf of stakeholders. To address the capacity issues identified in Sect. 1, Arcade supports early evaluation of partial requirements specifications, leveraging partitioned software architecture models along the two dimensions discussed in Sect. 2.2 (e.g., partitioning across the SEPA 3D Architecture models, and partitioning within a single SEPA 3D Architecture model).

Section 3.1 discusses evaluating correctness properties of the eDesign DRA. Section 3.2 discusses performance evaluation, and Sect. 3.3 discusses reliability evaluation. Section 3.4 presents the incremental approach taken for eDesign correctness evaluations, emphasising how subset models of the eDesign DRA helped mitigate complexity encountered while identifying and repairing correctness errors.

3.1 DRA correctness evaluation

Arcade employs the SPIN model checker and its associated Promela modeling language for evaluating three correctness properties of a DRA: *safety*, *liveness*, and *completeness* [4]. Each DRAC is represented by a Promela process. Predicates in the DRAC (e.g., pre-/post-conditions) are modelled as guards that either block or

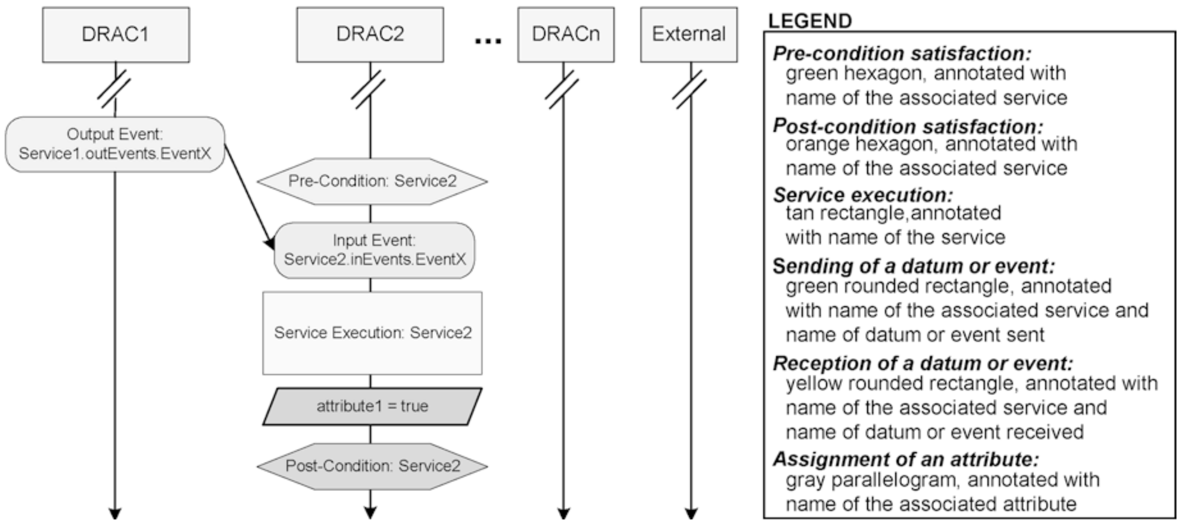

DRAC1
DRAC2
...
DRACn
External
Output Event: Service1.outEvents.EventX
Pre-Condition: Service2
Input Event: Service2.inEvents.EventX
Service Execution: Service2
attribute1 = true
Post-Condition: Service2
LEGEND
Pre-condition satisfaction: green hexagon, annotated with name of the associated service
Post-condition satisfaction: orange hexagon, annotated with name of the associated service
Service execution: tan rectangle,annotated with name of the service
Sending of a datum or event: green rounded rectangle, annotated with name of the associated service and name of datum or event sent
Reception of a datum or event: yellow rounded rectangle, annotated with name of the associated service and name of datum or event received
Assignment of an attribute: gray parallelogram, annotated with name of the associated attribute

Fig. 4 Architecture trace diagram description

enable other statements for execution. Triggers for pre-conditions and post-conditions include sending and reception of events and data. When a pre-condition becomes executable, the DRAC transitions to an "EXECUTING" state for the corresponding DRAC service. Eventually, the post-condition for the service becomes executable (assuming that the model is correct and the post-condition can trigger), and the DRAC returns to an "IDLE" state.

Model checking techniques require a model to be closed. Therefore, Arcade automatically generates two special Promela processes to model exchange of data and events with external entities: (1) the *Usage Profiles* process generates initial events/data required to initiate usage profile executions, and (2) the *External* process receives and generates all events/data that are designated as "external" to the DRA.

If a service has only data and event references in its pre-condition, these data and events will be generated by the *Usage Profiles* process upon SPIN startup. If a pre-condition has references to both internal and external input events/data, the *External* process will monitor for the existence of internal data/events and upon detecting them it will generate the external events/data identified in the pre-condition. The predicates for generating these events/data include a check to determine whether the external events/data are already in a channel. This check prevents false generation of duplicate events/data before the receiving service process has a chance to consume the data/events.

The following sections discuss each correctness property using examples from the eDesign case study.

3.1.1 *DRA safety evaluation*

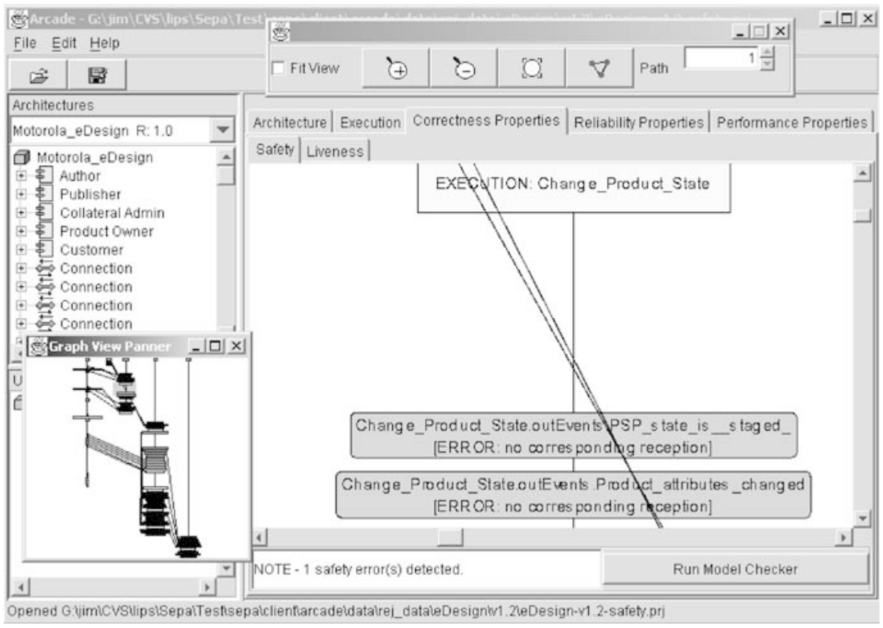
In the context of DRA evaluation, *safety* is informally defined as "the system never terminates in an undesirable end state" [15, 16, 17]. For a DRA, undesirable end states include unterminated service executions, and unconsumed events or data. Therefore, Arcade verifies that the following conditions hold: (1) if the pre-condition of a service has been satisfied, the service will eventually execute, (2) if a service executes, eventually its post-condition will be satisfied, and (3) all events and data produced as outputs of a service are eventually consumed. Formally (expressed in LTL):

$$\begin{aligned}\Box\,(\ &\forall \texttt{s:}\ \textit{Svc} \in \texttt{DRA} \mid \texttt{preCondition(s)} \rightarrow \Diamond\,\texttt{executed(s)} \rightarrow \Diamond\,\texttt{postCondition(s)}\ \wedge\\ &(\ \forall \texttt{e:}\ \textit{SvcOutEvt} \in \texttt{s.E}_o \mid \texttt{produced(e)} \rightarrow \Diamond\,\texttt{consumed(e)}\ )\ \wedge\\ &(\ \forall \texttt{d:}\ \textit{SvcOutData} \in \texttt{s.D}_o \mid \texttt{produced(d)} \rightarrow \Diamond\,\texttt{consumed(d)}\ )\ )\end{aligned}$$

where: $\texttt{Svc}$ is a DRA service; $\texttt{E}_o$ is the set of output events associated with a DRA service; and $\texttt{D}_o$ is the set of output data associated with a DRA service

SPIN produces counterexamples illustrating system executions in which safety errors occur, expressed in terms of the Promela model which has been model-checked by SPIN. For ease of understanding, Arcade presents SPIN counterexamples using an Architecture Trace Diagram (ATD) (Fig. 4) [18], an extension to ITU Message Sequence Charts [19]. In an ATD, the vertical dimension depicts time (top to bottom), and the horizontal dimension represents DRACs involved in the trace. Each DRAC has a lifeline that proceeds in the vertical dimension. Trace states are arranged in time sequence along the lifeline of their associated DRAC. Exchange of data and events in the DRA is shown by a directed arc from the sending state to the receiving state.

Figure 5 shows an Arcade ATD zoomed in to view safety errors discovered in the eDesign usage profile "Publish New Technical Document" ("ERROR: no corresponding reception"). The accompanying panner window shows a thumbnail image of the ATD. Safety errors were caused by two unconsumed events ("PSP state is staged", and "Product attributes changed") that have



EXECUTION: Change_Product_State

Change_Product_State.outEvents.PSP_state_is_staged_ [ERROR: no corresponding reception]

Change_Product_State.outEvents.Product_attributes_changed [ERROR: no corresponding reception]

NOTE - 1 safety error(s) detected.

Opened G:\jim\CVS\lips\Sepa\Test\sepa\client\arcade\data\rej_data\eDesign\v1.2\eDesign-v1.2-safety.prj

Fig. 5 Arcade presentation of safety errors (architecture trace diagram)

been output by the "Change Product State" service of the "Product Owner" DRAC (Fig. 3). These events were output as specified by the post-condition of the "Change Product State" service (Table 1), but were not subsequently consumed by any pre-condition. When this type of safety error occurs, the architect and domain experts must determine whether the errors occurred as a result of (1) an invalid post-condition (e.g. the "PSP state is changed" and "Product attributes changed" events should not have been generated in the post-condition for the "Change Product State" service), or (2) as a result of an invalid pre-condition associated with other service(s) of the DRA (e.g., some other service requires the "PSP state is staged" and "Product attributes changed" events, but this requirement was not specified in the DRA.) In this example the resolution was determined to be the latter (invalid pre-condition on another service). As a result, the eDesign DRA was updated to reflect the resolution.

3.1.2 *DRA liveness evaluation*

The liveness property for a DRA is informally defined as: "The system eventually enters all desirable states" [15, 16, 17]. A DRA has no liveness errors for this property when the following conditions hold: (1) no unreachable services exist, and (2) all required paths between services are traversable. Unreachable services occur in a DRA when an entire pre-condition is never satisfiable. Untraversable paths occur in a DRA when a disjunct sub-expression of a pre-condition is never satisfiable. Formally, this liveness property is defined by the following LTL expression:

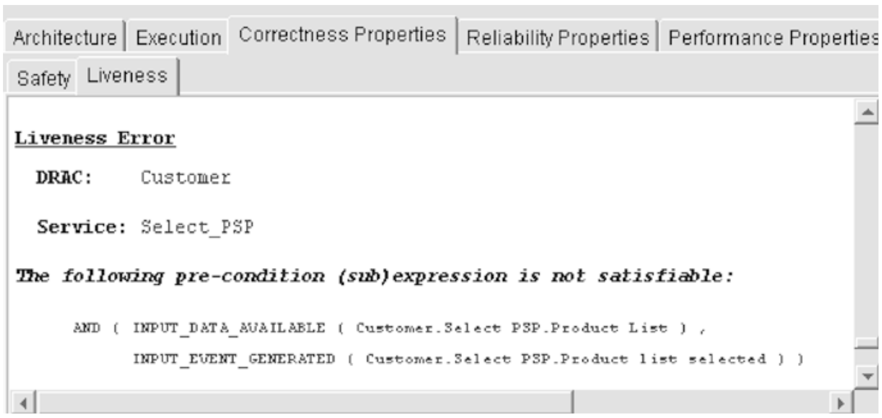
$\Box\,(\,\forall s\colon \mathit{Svc} \in \mathtt{DRA} \mid \forall x\colon \mathit{Expr} \in \mathtt{disjuncts}(s.C_{Pre}) \mid \Diamond x\,)$
where: $\mathtt{Svc}$ is a DRA service; and C_{Pre} is the DRA service precondition

Details of Promela code are difficult for stakeholders to interpret without specialised knowledge of SPIN/Promela (SPIN reports liveness errors as the specific line of Promela code that cannot be reached). Therefore, Arcade uses its knowledge of the DRA (and how it was translated to Promela) to present liveness evaluation results to stakeholders by indicating unsatisfiable pre-condition (sub)expressions in DRA terminology.

Figure 6 shows Arcade presenting an unsatisfiable pre-condition in the eDesign DRA. This liveness error occurred for the "Select PSP" service in the "Customer" DRAC. An eDesign domain expert determined that the "Product List" input data was never produced by any post-condition of any service, and the DRA was updated to correct the error.

3.1.3 *DRA completeness evaluation*

DRA completeness is informally defined as: "An architecture reflects the features required by domain experts." In other words, threads of execution specified within a DRA must reflect sequences of service executions that domain experts require (i.e., domain usage profiles), with no missing or extraneous service executions. Unfortunately, model checking tools such as SPIN cannot check that a software architecture provides correct semantics (completeness) without requiring additional properties to be defined in a language such as LTL [20]. This requirement actually *increases* the need for expertise when performing completeness evaluation with a model checker. Arcade bypasses this expertise issue by taking advantage of the following observations. Completeness errors are associated with unexpected behaviour of the system, where behaviors are described by respective usage profiles. These errors are typically manifested in sequences of service executions (usage profiles) that: (1) are missing expected service executions, (2) contain



Liveness Error

DRAC: Customer

Service: Select_PSP

The following pre-condition (sub)expression is not satisfiable:

```
AND ( INPUT_DATA_AVAILABLE ( Customer.Select PSP.Product List ) ,
      INPUT_EVENT_GENERATED ( Customer.Select PSP.Product list selected ) )
```

Fig. 6 Arcade presentation of liveness errors (unsatisfiable pre-condition)

unexpected service executions, (3) contain unexpected paths, or (4) are missing paths. Therefore, to assist the architect in verifying dynamic completeness, Arcade employs the SPIN guided simulation feature [14] to generate a visualisation of DRA threads of execution called an Execution Space (Fig. 7). Rather than specifying LTL expressions, domain experts can inspect an Execution Space to detect completeness errors. While this evaluation approach can be very complex for a large DRA, experience with eDesign indicates that the DRA subset model partitioning supported by Arcade can make this task manageable (to be discussed more in Sect. 3.1.4).

An Execution Space is a directed graph representing threads of service executions allowed by the DRA. A vertex in the graph represents a service execution state, and an edge represents a path from one service execution state to another. Two additional vertices in the Execution Space are a super-initial state (INIT), and a super-final state (TERM). These vertices provide common initiation and termination states for paths in the Execution Space.

Figure 7a illustrates an Execution Space for a subset model of the eDesign DRA (the "Access Product Information" usage profile). An eDesign domain expert identified a completeness error in this Execution Space, noting that there was no path from the "Search for Product" service to the TERM node. This path should exist to support the domain requirement to allow a customer to cancel their product search (Table 1). It was determined that the initial DRA was missing the "Request Cancelled" disjunct in the "Search for Product" post-condition. The corrected eDesign DRA produced the Execution Space in Fig. 7b.

Arcade creates an Execution Space via iteration over the following steps: (1) simulating DRA execution and merging all states occurring during simulation into a low-level state space (each simulation run creates a unique path in the low-level state space), and (2) performing a partial order reduction over the low-level state space with respect to service execution states. A path in the low-level state space is a total order representing one execution of the DRA. A branch in the low-level state space occurs when alternate interleavings of simulation events are possible (for example, different interleaving of sending or reception of events). Arcade performs partial order reduction of the low-level state space into the Execution Space by merging paths that share identical partial orders of service execution states.

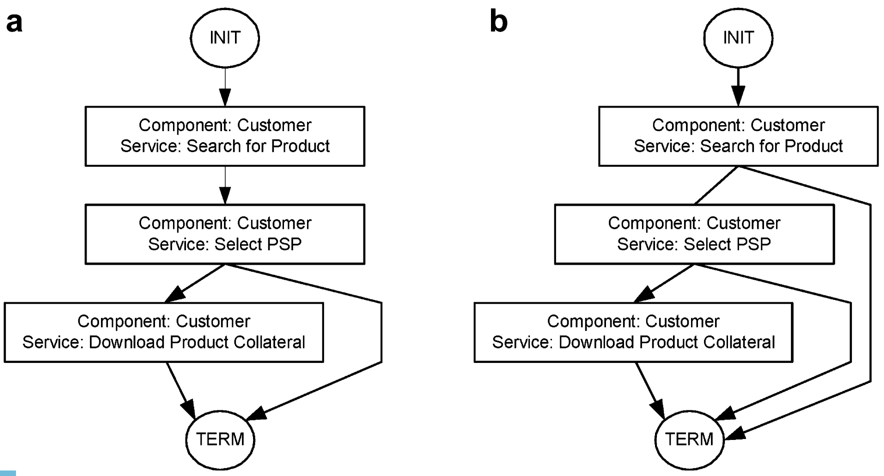


Fig. 7 **a** Execution space with completeness error. **b** Corrected execution space

3.1.4 *Using subset models to evaluate eDesign correctness*

To ensure that a system implementation meets stakeholder expectations, it is important to establish correctness of domain requirements specified in a DRA before using the DRA as a system blueprint. Furthermore, domain requirements must be correctly modelled before a cost-effective system targeting other non-functional requirements such as maintainability, performance, and reliability can be built [21, 1]. Correctness errors may arise as a result of a number of requirements acquisition and modelling issues, including (1) misstatement of requirements by stakeholders, (2) misrepresentation or misinterpretation of requirements by the requirements engineer, or (3) omission of requirements by stakeholders.

Arcade evaluation detected a number of correctness errors in the initial eDesign DRA. This was partially due to requirements acquisition and modeling issues such as those mentioned above, and partially due to the fact that correctness errors tend to propagate under model checking and simulation. For example, a liveness error associated with a particular service may cause many other liveness errors for other services that have causal dependencies on that service (e.g., the dependent services require events or data to become available via the post-condition associated with the service exhibiting the initial liveness error). Propagation of correctness errors makes it difficult for stakeholders to determine exactly which errors to address first. To mitigate this complexity issue, it was decided that correctness evaluations of DRA subset models would be appropriate for eDesign. This decision allowed evaluation to focus on requirements errors while minimising the effort expended to sort through correctness error propagations.

The technique employed to form DRA subset models was to further partition the requirements modelled in the initial DRA by slicing the DRA according to usage profile boundaries. This resulted in a number of DRA subset models, each of which was self-contained in terms of the ability to support execution of a single usage profile. The next steps were: (1) to iteratively evaluate and repair correctness errors for each of the subset models, (2) to merge the correct subset models into a unified DRA model, and (3) to evaluate the correctness of the resulting DRA.

Arriving at a correct eDesign DRA required five iterations of subset model evaluation. The number and types of errors detected, and their associated types of DRA corrections, are shown in Fig. 8. During the iterative evaluation and correction process, completeness errors were detected by stakeholder examinations of Execution Spaces produced by Arcade simulations, and safety and liveness errors were detected using Arcade's automated model checking.

Initially, completeness errors comprised the majority of errors uncovered by correctness evaluations. This first round of completeness errors was addressed primarily by adjusting pre- and post-conditions, and in one instance by distributing the domain functionality of a single DRA service across multiple services. Following these adjustments, safety and liveness errors began to be identified in greater numbers. The remaining iterations of evaluation and correction successively reduced the number of errors detected, and the majority of errors detected in the final iterations were safety errors. These results indicate that by the later iterations most of the required domain functionality was correctly modelled (e.g., there were fewer completeness and liveness errors), and there remained some adjustments to be made with regard to the details of data and event exchanges (e.g., most of the remaining errors were safety errors). Table 2 summarises the types of correctness errors found and the methods of detection for each revision of the eDesign DRA. These statistics highlight that simulation was an effective means of detecting correctness errors early in the process, and model checking was effective later in the process.

Arcade's incremental approach proved to be effective in support of detecting and repairing correctness errors. Following the final iteration of DRA subset model evaluation, the individual subset models were merged into a unified eDesign DRA, and correctness evaluation was reapplied. This final evaluation yielded no additional correctness errors, and at this point the team was ready to proceed with performance and reliability evaluation.

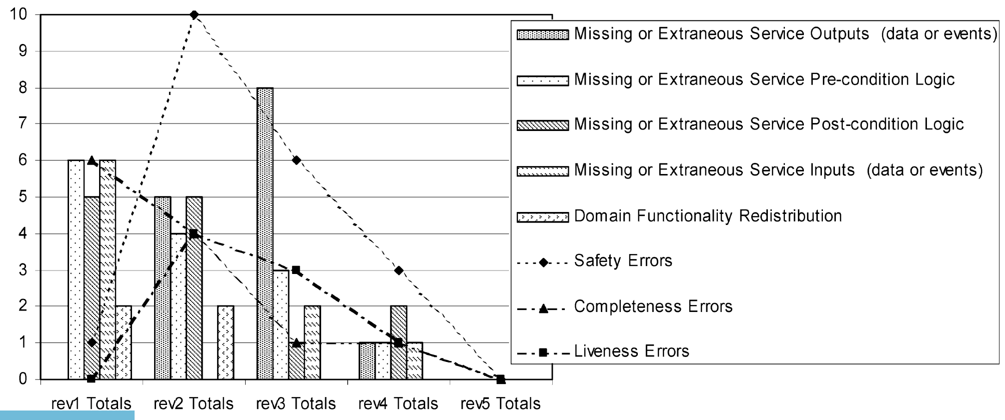


Fig. 8 eDesign correctness errors and resolutions

Table 2 Summary statistics for eDesign correctness evaluation

DRA revision	Safety errors	Liveness errors	Completeness errors	Detected by simulation	Detected model checking
1	1	0	6	6	1
2	10	4	4	4	14
3	6	3	1	1	9
4	3	1	1	1	4
5	0	0	0	0	0
Totals	20	8	12	12	28

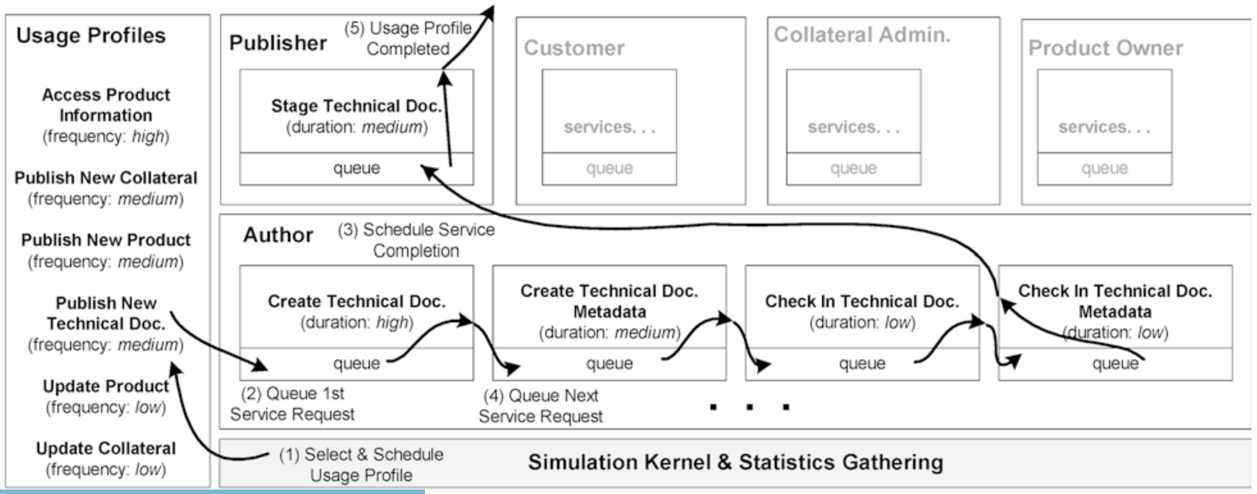
3.2 DRA performance evaluation

Arcade uses the Simpack simulation toolkit as the basis for its performance evaluations [22]. Simpack provides a simulation kernel and simulation support routines, including support for definition and management of resources to perform work (called facilities in Simpack), event scheduling and delivery, statistical distribution sampling, and statistics collection. Simulation with Simpack is performed by moving *tokens* (units of work) through *facilities* (work performers). Therefore, Arcade maps DRA services to Simpack facilities, and maps domain usage profiles to Simpack tokens. Arcade also defines simulation events and event handlers required to move usage profiles (tokens) through services (facilities) based on task sequencings specified in usage profiles. Low-level exchange of events and data are not modelled because these details are not necessary to perform the qualitative performance evaluations that Arcade delivers.

The approach is illustrated in Fig. 9 using the eDesign "Publish New Technical Document" usage profile (Table 1). Arcade initiates a simulation run by scheduling a *start_usage_profile* event using a negative exponential distribution to model the interarrival time of usage profile requests. When the Simpack kernel delivers a *start_usage_profile* event, Arcade randomly chooses a usage profile to begin executing (step 1 in Fig. 6). This choice is weighted by the frequencies of execution associated with each usage profile. When a usage profile has been selected, Arcade schedules (1) a *request_service* event for the first service in the selected usage profile, and (2) another *start_usage_profile* event (using the appropriate interarrival time). The *request_service* event is queued pending service availability (step 2 in Fig. 6). When the requested service becomes available, Arcade allocates the facility associated with the requested service (causing subsequent service requests to become queued waiting for service completion) and schedules a *service_completed* event in the simulation using a normal distribution centered around the service duration (step 3 in Fig. 6). When the *service_completed* event is delivered, Arcade schedules a *request_service* event for the next service defined in the usage profile (step 4 in Fig. 6). This process is repeated until the usage profile has been completed (step 5 in Fig. 6). Simulation continues in this manner until a maximum simulation time has been reached. Throughout this process, Arcade (with the aid of Simpack) collects performance statistics.

Arcade can produce many different performance measures for a DRA, including usage profile latency, service utilisation, and component utilisation [8]. Two eDesign performance measures are depicted in the graphs in Fig. 10: usage profile latencies, and service utilisation. Usage profile latency reflects the amount of time an eDesign customer can expect to wait for a usage profile to complete, and service utilisation indicates the amount of time a service spends performing work. The graphs in Fig. 10 can be correlated using their x-axes, where $IA\_<N>$ represents the mean interarrival time of *start_usage_profile* events (in N simulation units) over a set of simulation runs.

Fig. 9 Arcade performance simulation approach


Usage Profiles
Access Product Information (frequency: high)
Publish New Collateral (frequency: medium)
Publish New Product (frequency: medium)
Publish New Technical Doc. (frequency: medium)
Update Product (frequency: low)
Update Collateral (frequency: low)
Publisher
(5) Usage Profile Completed
Stage Technical Doc. (duration: medium)
queue
Customer
services. . .
queue
Collateral Admin.
services. . .
queue
Product Owner
services. . .
queue
Author
(3) Schedule Service Completion
Create Technical Doc. (duration: high)
queue
Create Technical Doc. Metadata (duration: medium)
queue
Check In Technical Doc. (duration: low)
queue
Check In Technical Doc. Metadata (duration: low)
queue
(2) Queue 1st Service Request
(4) Queue Next Service Request
(1) Select & Schedule Usage Profile
Simulation Kernel & Statistics Gathering

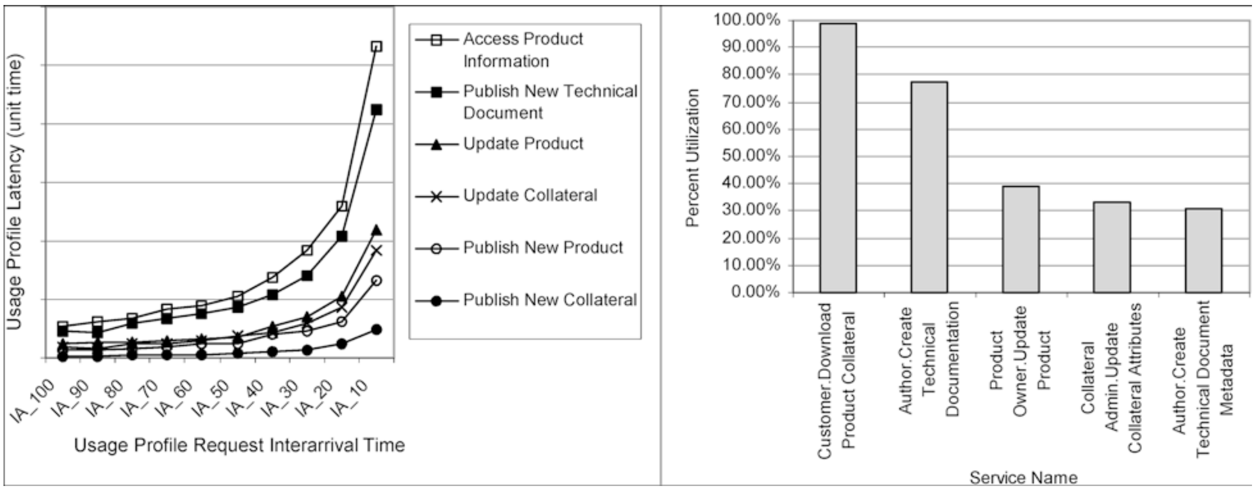


Fig. 10 eDesign usage profile latency and service utilization

According to eDesign domain experts, it is particularly important for the latency of the "Access Product Information" usage profile to be minimised in order to satisfy external customers. Noting that the "Download Product Collateral" service is a constituent of the "Access Product Information" usage profile (Table 1), the graphs in Fig. 10 indicate that the latency of the "Access Product Information" usage profile increases exponentially once the utilisation of the "Download Product Collateral" service reaches approximately 75% (at IA_45 when the mean interarrival time of *start_usage_profile* events is 45 simulation time units). The implication of this observation is that system implementers should seek to design the system in such a way that the "Download Product Collateral" service maintains utilisation of less than 75%.

DRA Performance concerns center on inherent domain performance characteristics, including: (1) exchange of events/data between services, (2) usage profiles that represent anticipated system usage patterns, (3) expected frequency of execution for usage profiles, and (4) expected service execution durations. Early performance evaluation using a DRA is useful in determining the performance characteristics realised by the domain requirements represented in the DRA. Later, when the AA and IA are specified, tradeoff decisions can be made with respect to knowledge gained from early evaluation of the DRA (e.g., for eDesign, critical attention should be paid to tradeoffs associated with choosing or building an application that provides the "Download Product Collateral" service, and when choosing the hardware and infrastructure on which this application is to be deployed). Once the AA and IA decisions have been made, additional information in the AA and IA can be supplied to Arcade for more detailed performance evaluations intended to assess the impact of these decisions on system performance. For example, recognising that the "Download Product Collateral" service was critical, eDesign stakeholders focused performance evaluations of the eDesign IA on understanding the CPU requirements for acceptable utilisation of the "Download Product Collateral Service" (with the intent of achieving an acceptable latency for the "Access Product Information" usage profile).

As requirements evolve, it is likely that evolution will occur more frequently for the AA and IA (Sect. 2.1). Therefore, when the DRA is reused, DRA evaluation results will still be valid with respect to domain requirements, and these results can also be reused to support the AA and IA evolution process. In addition, results of all the Arcade performance evaluations (DRA, AA, and IA) can serve as guidance to system implementers.

3.3 DRA reliability evaluation

Arcade employs a reliability estimation technique called Scenario-Based Reliability Analysis (SBRA) [23]. The SBRA technique consists of (1) constructing a probabilistic reliability model called a Component Dependency Graph (CDG), and (2) applying the SBRA algorithm to the CDG model to yield a reliability estimate. A CDG is constructed using an architecture model, estimated service execution times, and usage profiles with associated probabilities of execution. Arcade can perform reliability evaluation for individual services (SBRA-S), and for DRACs (SBRA-D) [6].

The CDG model used for SBRA-S is a connected graph defined by CDG = $<N, E, init, term>$, where N is the set of nodes in the graph, E is the set of edges in the graph, *init* is a common start node, and *term* is a common termination node. A node in the CDG represents a service, defined by $n_i = <SVC_i, R_{SVCi}, AE_{SVCi}>$, where SVC_i is the service name, R_{SVCi} is the reliability of SVC_i, and AE_{SVCi} is the average execution time of SVC_i. Directed edges in the CDG represent execution paths between services, and are defined as $e_{ij} = <T_{ij}, R_{Tij}, P_{Tij}>$ where T_{ij} is the transition name from node n_i to node n_j, R_{Tij} is the transition reliability, and P_{Tij} is the transition probability.

The SBRA algorithm calculates reliability by iterating over the transitions in a CDG graph model. During

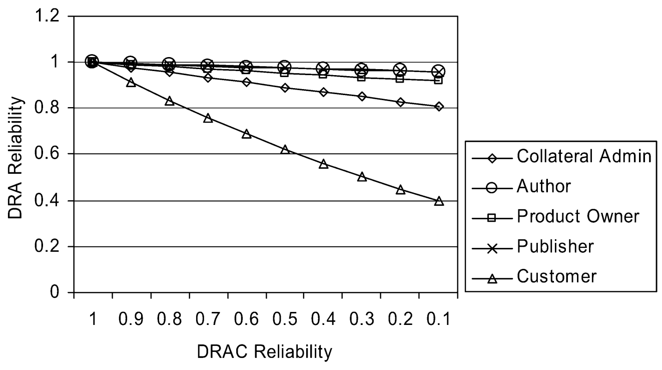


Fig. 11 Arcade reliability evaluation for eDesign DRA (SBRA-D)

iteration, the algorithm chooses paths based upon transition probabilities (P_{Tij}). As transitions are followed, cumulative reliability metrics are calculated using transition reliabilities (R_{Tij}) and service reliabilities (R_{SVCi}). Iteration continues until a pre-specified maximum system execution time is reached.

Arcade's approach for reliability evaluation using SBRA-S involves repeatedly applying the SBRA technique while (1) constraining the reliability of all services to 100% with the exception of a single service and (2) varying the reliability parameter of that service from 0% to 100% to assess the sensitivity of reliability of the entire system to the reliability of the service. This process is repeated for each service. The approach for DRA reliability evaluation using SBRA-D is similar to SBRA-S, with a focus on DRAC reliability rather than service reliability.

The SBRA-D evaluation results for the eDesign DRA are shown in Fig. 11. From these results it can be seen that the "Customer" DRAC has the most potential to impact system reliability. As with early DRA performance evaluation results, early DRA reliability results can serve to assist in AA and IA decision making, can be compared to the results of AA or IA reliability evaluations to assess the impact of decisions, and can provide guidance to system implementers.

4 Related work

With regard to correctness evaluation, many researchers recognise the need to work with early, partial requirements models. For example, researchers applying model checking to requirements of a fault tolerant system devoted considerable effort towards mitigating the size of the state space that would result from a model representing their requirements [19]. The approach selected was to partition requirements based upon the class of fault that would occur during system execution if a requirement were violated. A model of the system design was created, and requirements were expressed in LTL formulae and then model-checked. This approach focused on the important topic of mitigating complexity issues associated with memory and CPU utilisation, but did not address the issues of human expertise that are addressed by Arcade and the SEPA 3D Architecture (e.g., the issue of automatically creating the model to be checked from requirements specifications, and the issue of presenting results to non-expert stakeholders), nor was there a focus on early evaluation in support of requirements evolution and reuse. Because the DRA is a requirements specification, DRA evaluation can occur much early than checking whether a detailed system model can satisfy a specific requirement (or requirements set). Similarly, researchers using the Software Cost Reduction (SCR) approach acknowledged that presenting model checking results to users as logic formulae rather than in terms of the requirements models stakeholders were accustomed to remains an issue with their approach [24]. Other researchers have focused on compositional approaches to reduce complexity under model checking (e.g., partitioning by functional requirements encapsulation), but again the focus has not been on practicality issues associated with non-experts [25, 26]. An approach to partitioning requirements to be model checked based upon scenarios has been presented, but this work did not address presentation of model checking results to stakeholders in non-expert terminology [27].

In the performance evaluation domain, researchers have applied various approaches including queuing network models (QNM) [28, 29, 30], discrete event simulation [31, 32], Software Performance Engineering (SPE) [33], and process algebras [34] for performance evaluation of software architectures. Researchers working with QNM acknowledge a potential state explosion problem, and are seeking methods to reduce the model size based on MSC representations [35]. There is also a recognition that QNM based evaluation results must be translated into conclusions and recommendations for non-experts [29]. Researchers working with simulation approaches have recognised a need for automated translation of requirements into appropriate simulation models [3], and researchers working on process algebras have recognised that these formalisms need support for non-expert users (in this case via translation of architectural descriptions that integrate structural requirements with performance requirements into process algebra representations) [34]. While some of these research efforts recognise the need to support non-experts, and to allow partial model evaluation, these approaches do not explicitly integrate the requirements modelling activity with the performance evaluation activity as Arcade does, and most treat software architectures as design-level artifacts rather than requirements representations.

Researchers working on reliability evaluation have used simulation [36], analytical methods [37], and probabilistic models [22]. As with performance evaluation, most of this work does not explicitly recognise architectural models as requirements models, and the topics of partitioning requirements to mitigate complexity, promote requirements reuse, or help with requirements evolution do not appear (although Yacoub uses scenarios as the basis for evaluation, there is no discussion of partitioning of the model [23]). Furthermore, none of this research addresses mitigating the need for expertise in the selected technique, or the ability to support early evaluation.

5 Conclusions

This research investigates the efficacy of evaluating dynamic properties of requirements (e.g. *correctness*, *performance*, and *reliability*) early in the software lifecycle, when it is less costly to address errors. Efficacy issues include (1) practicality of performing evaluations and (2) usefulness of evaluation results (i.e., cost vs. benefit).

To mitigate a number of practical issues associated with dynamic property evaluation, Arcade leverages the SEPA 3D Architecture, a formal requirements representation that partitions requirements types amongst a set of interrelated architecture models. The eDesign case study illustrates how Arcade effectively uses the SEPA 3D Architecture to help manage complexity, to reduce the level of expertise required to perform dynamic property evaluations, and to support an iterative approach allowing early, incremental evaluation using partial models. While a small initial investment in time and training was required to adopt the Arcade approach, the eDesign team was subsequently able to gain valuable insight from correctness, performance, and reliability evaluations, for verifying requirements specified in the architecture and providing rationale for subsequent design and implementation decisions that would have otherwise been supported solely by intuition.

Arcade's incremental approach was effective for mitigating complexity while discovering correctness errors in the eDesign DRA. Correctness evaluation provided the opportunity to resolve functional errors (completeness, safety, and liveness) before significant architecture commitments had been made. Project participants acknowledged the cost savings in correcting errors in the DRA specification rather than correcting errors detected after system implementation.

From a performance and reliability perspective, Arcade's DRA evaluation yielded useful information early in the development process, highlighting critical architecture elements. These results influenced DRA refinement as well as subsequent design decisions involving application implementation and computing platform selection. For example, having determined the high utilisation of the service "Download Product Collateral" during DRA performance evaluations, subsequent eDesign performance evaluations focused on how non-functional and installation requirements affected this critical service. The eDesign Implementation Architecture (IA) specified an installation requirement to locate Content Delivery functionality (including the "Download Product Collateral" service) on a separate CPU from the Document Authoring functionality. The resulting Arcade IA performance evaluations suggested that Content Delivery functionality required a CPU with approximately four times the execution speed of the CPU supporting Document Authoring functionality.

A general observation of the eDesign case study is the value of early evaluation. Despite the DRA being a partitioned requirements representation focusing on domain functionality and data, early DRA evaluation results impacted subsequent architecting decisions. Under previously applied evaluation approaches, such errors were not caught until more detailed models (or the actual implementation) could be constructed (i.e., after significant design decisions were already made).

The basis for the Arcade approach is to reduce the need for tool expertise by providing a layer of automation between well-defined representations appropriate for early requirements specification and sophisticated evaluation techniques capable of offering valuable insight. However, as with any attempt to render an approach more easily adopted, reducing the need for expertise comes at the cost of limiting access to advanced features of underlying evaluation tools. For example, non-expert users who have no knowledge of model checkers and associated languages employed by Arcade (e.g., SPIN and Promela) can use Arcade to verify automatically defined correctness properties, but will not have the knowledge required to manually define additional properties for verification. While it is possible for an expert user to employ Arcade artifacts (for example the Promela code generated by Arcade) as a starting point for advanced evaluations using underlying evaluation tools directly, more research is needed to determine the extent to which advanced features can be made directly available to developers without requiring an unacceptable level of training and expertise, thereby reducing the tool's practical appeal.

Acknowledgements This research was supported in part by the Schlumberger Foundation Grant and the Texas Higher Education Coordinating Board Advanced Technology Program (ATP #003658-0188-1999).

References

1. Bass L, Clements P, Kazman R (1998) Software architecture in practice. SEI series in software engineering. Addison-Wesley, Reading, MA
2. Perry DE, Wolf AL (1992) Foundations for the study of software architecture. Softw Eng Notes 17(4):40–52
3. Hsia P, Davis A, Kung D (1993) Status report: requirements engineering. IEEE Softw 10(6):75–79
4. Wieringa R, Dubois E (1998) Integrating semi-formal and formal software specification techniques. Inform Syst 23(3/4):159–178
5. Barber KS et al (1999) Requirements evolution and reuse using the systems engineering process activities (SEPA). Aust J Inform Syst (Special Issue on Requirements Engineering) 7(1):75–97
6. Barber KS et al (2001) Reliability estimation techniques for domain reference architectures. In: 14th international conference on software and systems engineering and their applications (ICSSEA 2001), Paris

7. Barber KS, Graser TJ, Holt J. Evaluating dynamic correctness properties of domain reference architectures using a combination of simulation and model checking. In: 13th international conference in software engineering and knowledge engineering (SEKE 2001), Buenos Aires
8. Barber KS, Holt J, Baker G (2002) Performance evaluation of domain reference architectures. In: 14th international conference in software engineering and knowledge engineering (SEKE 2002), Ischia, Italy
9. Sommerville I (1992) Software engineering (4th edn). Addison-Wesley, Wokingham, UK
10. Tsai J, Xu K (1999) An empirical evaluation of deadlock detection in software architecture specifications. Ann Softw Eng 7:95–126
11. Hofmann HF, Lehner F (2001) Requirements engineering as a success factor in software projects. IEEE Softw 18(4):58–66
12. Barber KS, Graser TJ, Holt J (2001) Evolution of requirements and architectures: an empirical-based analysis. In: 1st international workshop on model-based requirements engineering (MBRE'01), San Diego, CA
13. Barber KS, Graser TJ, Holt J. A multi-level software architecture metamodel to support the capture and evaluation of stakeholder concerns. In: 5th world multi-conference on systematics, cybernatics and informatics (SCI 2001), Orlando, FL
14. Holzman GJ (1997) The model checker SPIN. IEEE Trans Softw Eng 23(5):279–295
15. Alpern B, Schneider FB (1987) Recognizing safety and liveness. Distrib Comput 2(3):117–126
16. Kindler E (1994) Safety and liveness properties: a survey. Bull Eur Assoc Theor Comput Sci 53:268–272
17. Lamport L, Lynch N (1990) Distributed computing: models and methods. In: Leeuwen Jv (ed) Handbook of theoretical computer science. Elsevier, Amsterdam, pp 1157–1199
18. Barber KS, Graser TJ, Holt J (2002) Providing early feedback in the development cycle through automated application of model checking to software architectures. In: 16th international conference on automated software engineering, San Diego, CA
19. ITU-TS (1996) ITU-TS Recommendation Z.120: Message Sequence Charts (MSC). ITU, Geneva
20. Schneider F et al (1998) Validating requirements for fault tolerant systems using model checking. In: 3rd international conference on requirements engineering, Colorado Springs, CO
21. Barber KS, Holt J (2001) Software architecture correctness. IEEE Softw 18(8):64–65
22. Fishwick PA (1995) Simulation model design and execution: building digital worlds. Prentice-Hall, Englewood Cliffs, NJ
23. Yacoub S, Cukic B, Ammar H (1999) Scenario-based reliability analysis of component-based software. In: 10th international symposium on software reliability engineering, Boca Raton, FL
24. Heitmeyer C, Kirby J, Labaw B (1998) Applying the SCR requirements method to a weapons control panel: an experience report. In: 2nd workshop on formal methods in software practice (FMSP'98), Clearwater Beach, FL
25. Cheung S, Giannakopoulou D, Kramer J (1997) Verification of liveness properties using compositional reachability analysis. In: ESEC/FSE '97, Zurich
26. Cheung SC, Kramer J Checking subsystem safety properties in compositional reachability analysis. In: 18th international conference on software engineering, Berlin
27. Bose P (1999) Scenario-driven analysis of component-based software architecture models. In: IFIP WICSA, San Antonio, TX
28. Aquilana F, Balsamo S, Inverardi P (2001) Performance analysis at the software architectural design level. Perform Evaluation 45(2–3):147–178
29. Petriu D, Shousha C, Jalnapurkar A (2000) Architecture-based performance analysis applied to a telecommunication system. IEEE Trans Softw Eng 26(11):1049–1065
30. Spitznagel B, Garlan D (1998) Architecture-based performance analysis. In: 10th international conference on software engineering and knowledge engineering, San Francisco, CA
31. Li JJ (1998) Performance prediction based on semi-formal software architectural description. In: International conference on performance in computing and communications, Phoenix/Tempe, AZ
32. Lung C-H, Jalnapurkar A, El-Rayess A (1998) Performance-oriented software architecture engineering: an experience report. In: Workshop on software performance (WOSP98), Santa Fe, NM
33. Williams LG, Smith CU (1998) Performance evaluation of software architectures. In: Workshop on software and performance, Santa Fe, NM
34. Bernardo M, Ciancarini P, Donatiello L (2000) AEMPA: a process algebraic description language for the performance analysis of software architectures. In: 2nd international workshop on software and performance (WOSP 2000), Ottawa
35. Andolfi F et al (2000) Deriving performance models of software architecture from message sequence charts. In: 2nd international workshop on software performance, Ottawa
36. Li JJ, Micallef J (1997) Automatic simulation to predict software architecture reliability. In: 8th international symposium on software reliability engineering, Albuquerque, NM
37. Gokhale SS et al (1998) An analytical approach to architecture-based software reliability prediction. In: IEEE international computer performance and dependability symposium, Durham, NC