Domains

Their Simulation, Monitoring and Control

— A Divertimento of Ideas and Suggestions —

Dines Bjørner[1]
[1] Fredsvej 11, DK-2840 Holte, Denmark.
Techn.Univ. of Denmark, DK-2800 Kgs. Lyngby, Denmark.
e-mail: bjorner@gmail.com, URL: www.imm.dtu.dk/~dibj

Written in 2008, compiled: September 24, 2018: 08:51 am

Abstract

We sketch some observations of the concepts of domain, requirements and modeling – where abstract interpretations of these models cover both a priori, a posteriori and real-time aspects of the domain as well as 1–1 (i.e., real-time), microscopic and macroscopic simulations, real-time monitoring and real-time monitoring & control of that domain. The reference frame for these concepts are domain models: carefully narrated and formally described domains. On the basis of a familiarising [Appendix **??**] example of a domain description, we survey more-or-less standard ideas of verifiable software developments and conjecture software product families of demos, simulators, monitors and monitors & controllers – but now these "standard ideas" are recast in the context of core requirements prescriptions being "derived" from domain descriptions.

Contents

1 Introduction

A background setting for this paper is the concern for (α) professionally developing the right software, i.e., software which satisfies users expectations, and (ω) software that is right: i.e., software which is correct with respect to user requirements and thus has no "bugs", no "blue screens". The present paper must be seen on the background of a main line of experimental research around the topics of domain science & engineering and requirements engineering and their relation. For details I refer to [Bjø16b, Bjø18, Bjø16a].

"Confusing Demos": This author has had the doubtful honour, on his many visits to computer science and software engineering laboratories around the world, to be presented, by his colleagues' aspiring PhD students, so-called demos of "systems" that they were investigating. There always was a tacit assumption, namely that the audience, i.e., me, knew, a priori, what the domain "behind" the "system" being "demo'ed" was. Certainly, if there was such an understanding, it was brutally demolished by the "demo" presentation. My questions, such as *"what are you demo'ing"* (etcetera) went unanswered. Instead, while we were waiting to see "something interesting" to be displayed on the computer screen we were witnessing frantic, sometimes failed, input of commands and data,

"nervous" attempts with "mouse" clickings, etc. – before something intended was displayed. After a, usually 15 minute, grace period, it was time, luckily, to proceed to the next "demo".

Aims & Objectives: The aims of this paper is to present (a) some ideas about software that either "demo", simulate, monitor or monitor & control domains; (b) some ideas about "time scaling": demo and simulation time versus domain time; and (c) how these kinds of software relate. The (undoubtedly very naïve) objectives of the paper is also to improve the kind of demo-presentations, alluded to above, so as to ensure that the basis for such demos is crystal clear from the very outset of research & development, i.e., that domains be well-described. The paper, we think, tackles the issue of so-called 'model-oriented (or model-based) software development' from altogether different angles than usually promoted.

An Exploratory Paper: The paper is exploratory. There will be no theorems and therefore there will be no proofs. We are presenting what might eventually emerge into (α) a theory of domains, i.e., a domain science [Bjø07, BE10, Bjø09a, Bjø11b], and (β) a software development theory of domain engineering versus requirements engineering [Bjø11a, Bjø08, Bjø09b, Bjø10b].

The paper is not a "standard" research paper: it does not compare its claimed achievements with corresponding or related achievements of other researchers – simply because we do not claim "achievements" which have been reasonably well formalised. But we would suggest that you might find some of the ideas of the paper (in Sect. 3) worthwhile. Hence the "divertimento" suffix to the paper title.

Structure of Paper: The structure of the paper is as follows. In Appendix **??** we present a fair-sized example of a domain description. In Sect. 3 we then outline a series of interpretations of domain descriptions. These arise, when developed in an orderly, professional manner, from requirements prescriptions which are themselves orderly developed from the domain description[1], cf. [Bjø16a].

The essence of Sect. 3 is (i) the (albeit informal) presentation of such tightly related notions as *demos* (Sect. 3.1), *simulators* (Sect. 3.2), *monitors* (Sect. 3.3.1) and *monitors & controllers* (Sect. 3.3.2) (these notions can be formalised), and (ii) the conjectures on a product family of domain-based software developments (Sect. 3.5). A notion of *script-based simulation* extends demos and is the basis for monitor and controller developments and uses. The scripts used in our examples are related to time, but one can define non-temporal scripts – so the "carrying idea" of Sect. 3 extends to a widest variety of software. We claim that Sect. 3 thus brings these new ideas: a tightly related software engineering concept of *demo-simulator-monitor-controller* machines, and an extended notion of *reference models for requirements and specifications* [GGJZ00].

2 Domain Descriptions

By a domain description we shall mean a combined narrative, that is, precise, but informal, and a formal description of the application domain ***as it is:*** no reference to any possible requirements let alone software that is desired for that domain. Thus a requirements prescription is a likewise combined precise, but informal, narrative, and a formal prescription of what we expect from a machine (hardware + software) that is to support endurants, actions, events and behaviours of a possibly business process re-engineered application domain. Requirements expresses a domain ***as we would like to to be.***

We present an example domain description in Appendix **??**.

[1] We do not show such orderly "derivations" but outline their basics in Sect. 3.4.2.

We further refer to the literature for examples: [Bjø00, *railways* (2000)], [Bjø02, *the 'market'* (2000)], [Bjø09b, *public government, IT security, hospitals* (2006) chapters 8–10], [Bjø08, *transport nets* (2008)] and [Bjø10b, *pipelines* (2010)]. On the net you may find technical reports covering "larger" domain descriptions. "Older" publications on the concept of domain descriptions are [Bjø10b, Bjø11b, Bjø09c, BE10, Bjø08, Bjø07, Bjø10a] all summarised in [Bjø16b, Bjø18, Bjø16a].

Domain descriptions do not necessarily describe computable objects. They relate to the described domain in a way similar to the way in which mathematical descriptions of physical phenomena stand to "the physical world".

3 Interpretations

In this main section of the paper we present a number of interpretations of rôles of domain descriptions.

3.1 What Is a Domain-based Demo?

A *domain-based demo* is a software system which *"present"* endurants and perdurants[2]: actions, events and behaviours of a domain. The *"presentation"* abstracts these phenomena and their related concepts in various computer generated forms: visual, acoustic, etc.

3.1.1 Examples

There are two main examples. One is given in Appendix **??**. The other is summarised below. It is from our paper on "deriving requirements prescriptions from domain descriptions" [Bjø16a]. The summary follows.

The domain description of Sect. 2. of [Bjø16a], outlines an abstract concept of transport nets (of hubs [street intersections, train stations, harbours, airports] and links [road segments, rail tracks, shipping lanes, air-lanes]), their development, traffic [of vehicles, trains, ships and aircraft], etc. We shall assume such a transport domain description below.

Endurants are, for example, presented as follows: (a) transport nets by two dimensional (2D) road, railway or air traffic maps, (b) hubs and links by highlighting parts of 2D maps and by related photos – and their unique identifiers by labeling hubs and links, (c) routes by highlighting sequences of paths (hubs and links) on a 2D map, (d) buses by photographs and by dots at hubs or on links of a 2D map, and (e) bus timetables by, well, indeed, by showing a 2D bus timetable.

Actions are, for example, presented as follows: (f) The insertion or removal of a hub or a link by showing "instantaneous" triplets of "before", "during" and "after" animation sequences. (g) The start or end of a bus ride by showing flashing animations of the appearance, respectively the flashing disappearance of a bus (dot) at the origin, respectively the destination bus stops.

Events are, for example, presented as follows: (h) A mudslide [or fire in a road tunnel, or collapse of a bridge] along a (road) link by showing an animation of part of a (road) map with an instantaneous sequence of (α) the present link , (β) a gap somewhere on the link, (γ) and the appearance of two ("symbolic") hubs "on either side of the gap". (i) The congestion of road traffic "grinding to a halt" at, for example, a hub, by showing an animation of part of a (road) map with an instantaneous sequence of the massive accumulation of vehicle dots moving (instantaneously) from two or more links into a hub.

Behaviours are, for example, presented as follows: (k) A bus tour: from its start, on time, or "thereabouts", from its bus stop of origin, via (all) intermediate stops, with or without delays or advances in times of arrivals and departures, to the bus stop of destination (ℓ) The composite behaviour

[2]The concepts of 'endurants' and 'perdurants' were defined in [Bjø16b].

of "all bus tours", meeting or missing connection times, with sporadic delays, with cancellation of some bus tours, etc. – by showing the sequence of states of all the buses on the net.

We say that behaviours $((j)$–$(\ell))$ are *script-based* in that they (try to) satisfy a bus timetable $((e))$.

3.1.2 Towards a Theory of Visualisation and Acoustic Manifestation

The above examples shall serve to highlight the general problem of visualisation and acoustic manifestation. Just as we need sciences of visualising scientific data and of diagrammatic logics, so *we need more serious studies of visualisation and acoustic manifestation — so amply, but, this author thinks, inconsistently demonstrated by current uses of interactive computing media.*

3.2 Simulations

"Simulation is the imitation of some real thing, state of affairs, or process; the act of simulating something generally entails representing certain key characteristics or behaviours of a selected physical or abstract system" [Wikipedia] for the purposes of testing some hypotheses usually stated in terms of the model being simulated and pairs of statistical data and expected outcomes.

3.2.1 Explication of Figure 1

Figure 1 on the following page attempts to indicate four things: (i) Left top: the rounded edge rectangle labeled "The Domain" alludes to some specific domain ("out there"). (ii) Left middle: the small rounded rectangle labeled "A Domain Description" alludes to some document which narrates and formalises a description of "the domain". (iii) Left bottom: the medium sized rectangle labeled "A Domain Demo based on the Domain Description" (for short "Demo") alludes to a software system that, in some sense (to be made clear later) "simulates" "The Domain." (iv) Right: the large rectangle (a) shows a horisontal time axis which basically "divides" that large rectangle into two parts: (b) Above the time axis the **"fat"** rounded edge rectangle alludes to the time-wise behaviour, a *domain trace*, of "The Domain" (i.e., the actual, the real, domain). (c) Below the time axis there are eight **"thin"** rectangles. These are labels S1, S2, S3, S4, S5, S6, S7 and S8. (d) Each of these denote a "run", i.e., a time-stamped "execution", a *program trace*, of the "Demo". Their "relationship" to the time axis is this: their execution takes place in the real time as related to that of "The Domain" behaviour.

A *trace* (whether a domain or a program execution trace) is a time-stamped sequence of states: domain states, respectively demo, simulator, monitor and monitor & control states.

From Fig. 1 on the next page and the above explication we can conclude that "executions" S4 and S5 each share exactly one time point, t, at which "The Domain" and "The Simulation" "share" time, that is, the time-stamped execution S4 and S5 reflect a "Simulation" state which at time t should reflect (some abstraction of) "The Domain" state.

Only if the domain behaviour (i.e., trace) fully "surrounds" that of the simulation trace, or, vice-versa (cf. Fig. 1[S4,S5]), is there a "shared" time. Only if the 'begin' and 'end' times of the domain behaviour are identical to the 'start' and 'finish' times of the simulation trace, is there an infinity of shared 1–1 times. Only then do we speak of a real-time simulation.

In Fig 2 on page 8 we show "the same" "Domain Behaviour" (three times) and a (1) simulation, a (2) monitoring and a (3) monitoring & control, all of whose 'begin/start' (b/β) and 'end/finish' (e/ϵ) times coincide. In such cases the "Demo/Simulation" takes place in real-time throughout the 'begin$\cdots\cdots$end' interval.

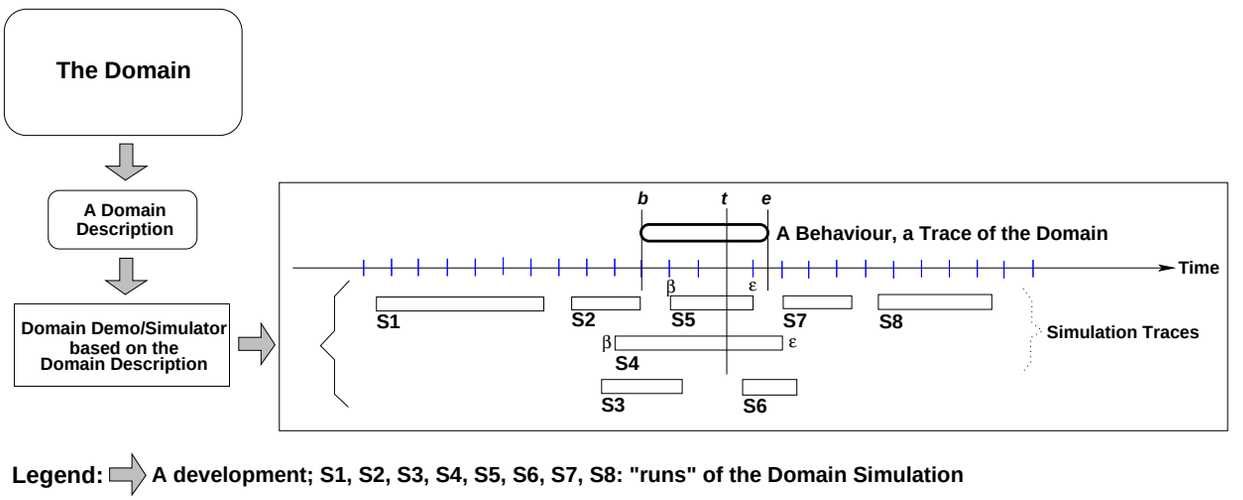


Figure 1: Simulations

Let β and ϵ be the 'start' and 'finish' times of either S4 or S5. Then the relationship between t, β, ϵ, b and e is $\frac{t-\mathsf{b}}{\mathsf{e}-t} = \frac{t-\beta}{\epsilon-t}$ — which leads to a second degree polynomial in t which can then be solved in the usual, high school manner.

3.2.2 Script-based Simulation

A script-based simulation is the behaviour, i.e., an execution, of, basically, a demo which, step-by-step, follows a script: that is a prescription for highlighting endurants, actions, events and behaviours.

Script-based simulations where the script embodies a notion of time, like a bus timetable, and unlike a route, can be thought of as the execution of a demos where "chunks" of demo operations take place in accordance with "chunks"[3] of script prescriptions. The latter (i.e., the script prescriptions) can be said to represent simulated (i.e., domain) time in contrast to "actual computer" time. The actual times in which the script-based simulation takes place relate to domain times as shown in Simulations S1 to S8 in Fig. 1 and in Fig. 2(1–3). Traces Fig. 2(1–3) and S8 Fig. 1 are said to be *real-time:* there is a one-to-one mapping between computer time and domain time. S1 and S4 Fig. 1 are said to be *microscopic:* disjoint computer time intervals map into distinct domain times. S2, S3, S5, S6 and S7 are said to be *macroscopic:* disjoint domain time intervals map into distinct computer times.

In order to concretise the above "vague" statements let us take the example of simulating bus traffic as based on a bus timetable script. A simulation scenario could be as follows. Initially, not relating to any domain time, the simulation "demos" a net, available buses and a bus timetable. The person(s) who are requesting the simulation are asked to decide on the ratio of the domain time interval to simulation time interval. If the ratio is 1 a real-time simulation has been requested. If the ratio is less than 1 a microscopic simulation has been requested. If the ratio is larger than 1 a microscopic simulation has been requested. A chosen ratio of, say 48 to 1 means that a 24 hour bus traffic is to be simulated in 30 minutes of elapsed simulation time. Then the person(s) who are requesting the simulation are asked to decide on the starting domain time, say 6:00am, and the domain time interval of simulation, say 4 hours – in which case the simulation of bus traffic from 6am

[3] We deliberately leave the notion of chunk vague so as to allow as wide an spectrum of simulations.

till 10am is to be shown in 5 minutes (300 seconds) of elapsed simulation time. The person(s) who are requesting the simulation are then asked to decide on the *"sampling times"* or *"time intervals"*: If *'sampling times'* 6:00 am, 6:30 am, 7:00 am, 8:00 am, 9:00 am, 9:30 am and 10:00 am are chosen, then the simulation is stopped at corresponding simulation times: 0 sec., 37.5 sec., 75 sec., 150 sec., 225 sec., 262.5 sec. and 300 sec. The simulation then shows the state of selected endurants and actions at these domain times. If *'sampling time interval'* is chosen and is set to every 5 min., then the simulation shows the state of selected endurants and actions at corresponding domain times. The simulation is resumed when the person(s) who are requesting the simulation so indicates, say by a "resume" icon click. The time interval between adjacent simulation stops and resumptions contribute with 0 time to elapsed simulation time – which in this case was set to 5 minutes. Finally the requestor provides some statistical data such as numbers of potential and actual bus passengers, etc.

Then two clocks are started: a domain time clock and a simulation time clock. The simulation proceeds as driven by, in this case, the bus time table. To include "unforeseen" events, such as the wreckage of a bus (which is then unable to complete a bus tour), we allow any number of such events to be randomly scheduled. Actually scheduled events "interrupts" the "programmed" simulation and leads to thus unscheduled stops (and resumptions) where the unscheduled stop now focuses on showing the event.

3.2.3 The Development Arrow

The arrow, ⇨, between a pair of boxes (of Fig. 1 on the previous page) denote a step of development: (i) from the domain box to the domain description box, ⇩, it denotes the development of a domain description based on studies and analyses of the domain; (ii) from the domain description box to the domain demo box, ⇩, it denotes the development of a software system — where that development assumes an intermediate requirements box which has not been show; (iii) from the domain demo box to either of a simulation traces, ⇨, it denotes the development of a simulator as the related demo software system, again depending on whichever special requirements have been put to the simulator.

3.3 Monitoring & Control

Figure 2 on the following page shows three different kinds of uses of software systems (where (2) [Monitoring] and (3) [Monitoring & Control] represent further) developments from the demo or simulation software system mentioned in Sect. 3.1 and Sect. 3.2.2 on the previous page. We have added some (three) horisontal and labeled (p, q and r) lines to Fig. 2 on the following page(1,2,3) (with respect to the traces of Fig. 1 on the previous page). They each denote a trace of a endurant, an action or an event, that is, they are traces of values of these phenomena or concepts. A (named) endurant value entails a description of the endurant, whither atomic ('hub', 'link', 'bus timetable') or composite ('net', 'set of hubs', etc.): of its unique identity, its mereology and a selection of its attributes. A (named) action value could, for example, be the pair of the before and after states of the action and some description of the function ('insertion of a link', 'start of a bus tour') involved in the action. A (named) event value could, for example, be a pair of the before and after states of the endurants causing, respectively being effected by the event and some description of the predicate ('mudslide', 'break-down of a bus') involved in the event. A cross section, such as designated by the vertical lines (one for the domain trace, one for the "corresponding" program trace) of Fig. 2 on the following page(1) denotes a state: a domain, respectively a program state.

Figure 2(1) attempts to show a real-time demo or simulation for the chosen domain. Figure 2(2) purports to show the deployment of real-time software for monitoring (chosen aspects of) the chosen domain. Figure 2(3) purports to show the deployment of real-time software for monitoring as well as controlling (chosen aspects of) the chosen domain.

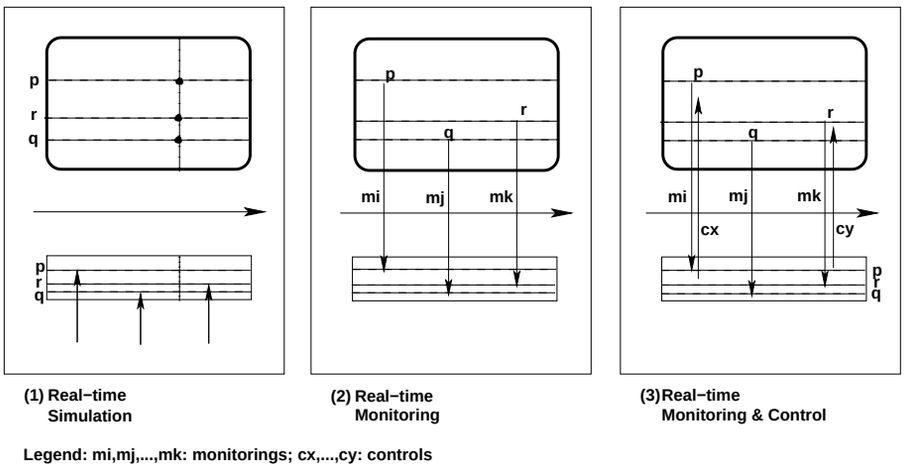


Figure 2: Simulation, Monitoring and Monitoring & Control

3.3.1 Monitoring

By *domain monitoring* we mean *"to be aware of the state of a domain"*, its endurants, actions, events and behaviour. Domain monitoring is thus a process, typically within a distributed system for collecting and storing state data. In this process "observation" points — i.e., endurants, actions and where events may occur — are identified in the domain, cf. points p, q and r of Fig. 2. Sensors are inserted at these points. The "downward" pointing vertical arrows of Figs. 2(2–3), from "the domain behaviour" to the "monitoring" and the "monitoring & control" traces express communication of what has been sensed (measured, photographed, etc.) [as directed by and] as input data (etc.) to these monitors. The monitor (being "executed") may store these "sensings" for future analysis.

3.3.2 Control

By *domain control* we mean *"the ability to change the value"* of endurants and the course of actions and hence behaviours, including prevention of events of the domain. Domain control is thus based on domain monitoring. Actuators are inserted in the domain "at or near" monitoring points or at points related to these, viz. points p and r of Fig. 2(3). The "upward" pointing vertical arrows of Fig. 2(3), from the "monitoring & control" traces to the "domain behaviour" express communication, to the domain, of what has been computed by the controller as a proper control reaction in response to the monitoring.

3.4 Machine Development

3.4.1 Machines

By a *machine* we shall understand a combination of hardware and software. For demos and simulators the machine is "mostly" software with the hardware typically being graphic display units with tactile instruments. For monitors the "main" machine, besides the hardware and software of demos and simulators, additionally includes *sensors* distributed throughout the domain and the technological machine means of *communicating* monitored signals from the sensors to the "main" machine and the processing of these signals by the main machine. For monitors & controllers the machine, besides

the monitor machine, further includes actuators placed in the domain and the machine means of computing and communicating control signals to the actuators.

3.4.2 Requirements Development

Essential parts of ℝequirements to a 𝕄achine can be systematically "derived" from a 𝔻omain description. These essential parts are the *domain requirements* and the *interface requirements*. Domain requirements are those requirements which can be expressed, say in narrative form, by mentioning technical terms only of the domain. These technical terms cover only phenomena and concepts (endurants, actions, events and behaviours) of the domain. Some domain requirements are *projected*, *instantiated*, made more *deterministic* and *extended*[4]. We bring examples that are taken from Sect. 2. of [Bjø16a], cf. Sect. 3.1.1 on page 4 of this paper. (a) By *domain projection* we mean a sub-setting of the domain description: parts are left out which the requirements stake-holders, collaborating with the requirements engineer, decide is of no relevance to the requirements. For our example it could be that our domain description had contained models of road net attributes such as "the wear & tear" of road surfaces, the length of links, states of hubs and links (that is, [dis]allowable directions of traffic through hubs and along links), etc. Projection might then omit these attributes. (b) By *domain instantiation* we mean a specialisation of endurants, actions, events and behaviours, refining them from abstract simple entities to more concrete such, etc. For our example it could be that we only model freeways or only model road-pricing nets – or any one or more other aspects. (c) By *domain determination* we mean that of making the domain description cum domain requirements prescription less non-deterministic, i.e., more deterministic (or even the other way around!). For our example it could be that we had domain-described states of street intersections as not controlled by traffic signals – where the determination is now that of introducing an abstract notion of traffic signals which allow only certain states (of red, yellow and green). (d) By *domain extension* we basically mean that of extending the domain with phenomena and concepts that were not feasible without information technology. For our examples we could extend the domain with bus mounted GPS gadgets that record and communicate (to, say a central bus traffic computer) the more-or-less exact positions of buses – thereby enabling the observation of bus traffic. Interface requirements are those requirements which can be expressed, say in narrative form, by mentioning technical terms both of the domain and of the machine. These technical terms thus cover shared phenomena and concepts, that is, phenomena and concepts of the domain which are, in some sense, also (to be) represented by the machine. Interface requirements represent (i) the initialisation and "on-the-fly" update of machine endurants on the basis of *shared* domain endurants; (ii) the interaction between the machine and the domain while the machine is carrying out a (previous domain) action; (iii) machine responses, if any, to domain events — or domain responses, if any, to machine events cum "outputs"; and (iv) machine monitoring and machine control of domain phenomena. Each of these four (i–iv) interface requirement facets themselves involve projection, instantiation, determination, extension and fitting. Machine requirements are those requirements which can be expressed, say in narrative form, by mentioning technical terms only of the machine. (An example is: visual display units.)

3.5 Verifiable Software Development

3.5.1 An Example Set of Conjectures

We illustrate some conjectures.

(A) From a domain, $\mathcal{D}$, one can develop a domain description $\mathbb{D}$. $\mathbb{D}$ cannot be [formally] verified. It can be [informally] validated "against" $\mathcal{D}$. Individual properties, $\mathbb{P}_{\mathbb{D}}$, of the domain description $\mathbb{D}$

[4] We omit consideration of *fitting*.

and hence, purportedly, of the domain, $\mathcal{D}$, can be expressed and possibly proved $\mathbb{D} \models \mathbb{P}_{\mathbb{D}}$ and these may be validated to be properties of $\mathcal{D}$ by observations in (or of) that domain.

(B) From a domain description, $\mathbb{D}$, one can develop requirements, $\mathbb{R}_{\text{DE}}$, for, and from $\mathbb{R}_{\text{DE}}$ one can develop a domain demo machine specification $\mathbb{M}_{\text{DE}}$ such that $\mathbb{D}, \mathbb{M}_{\text{DE}} \models \mathbb{R}_{\text{DE}}$. The formula $\mathbb{D}, \mathbb{M} \models \mathbb{R}$ can be read as follows: in order to prove that the Machine satisfies the Requirements, assumptions about the Domain must often be made explicit in steps of the proof.

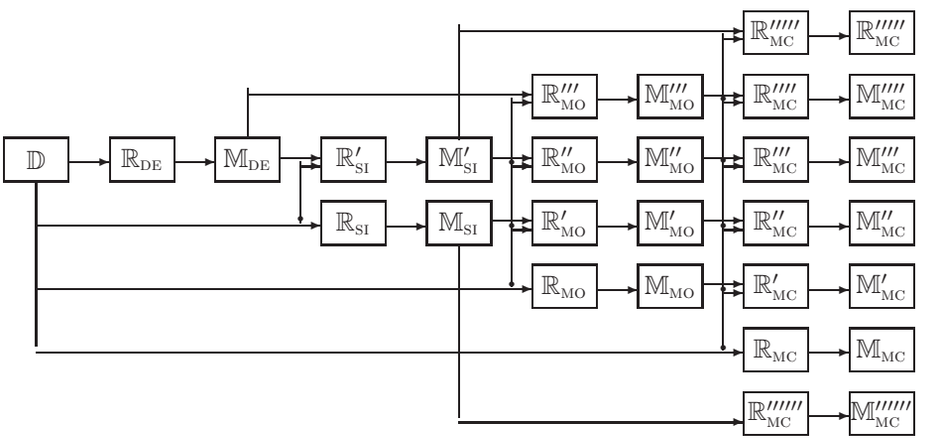
(C) From a domain description, $\mathbb{D}$, and a domain demo machine specification, $\mathbb{S}_{\text{DE}}$, one can develop requirements, $\mathbb{R}_{\text{SI}}$, for, and from such a $\mathbb{R}_{\text{SI}}$ one can develop a domain simulator machine specification $\mathbb{M}_{\text{SI}}$ such that $(\mathbb{D}; \mathbb{M}_{\text{DE}}), \mathbb{M}_{\text{SI}} \models \mathbb{R}_{\text{SI}}$. We have "lumped" $(\mathbb{D}; \mathbb{M}_{\text{DE}})$ as the two constitute the extended domain for which we, in this case of development, suggest the next stage requirements and machine development to take place.

(D) From a domain description, $\mathbb{D}$, and a domain simulator machine specification, $\mathbb{M}_{\text{SI}}$, one can develop requirements, $\mathbb{R}_{\text{MO}}$, for, and from such a $\mathbb{R}_{\text{MO}}$ one can develop a domain monitor machine specification $\mathbb{M}_{\text{MO}}$ such that $(\mathbb{D}; \mathbb{M}_{\text{SI}}), \mathbb{M}_{\text{MO}} \models \mathbb{R}_{\text{MO}}$.

(E) From a domain description, $\mathbb{D}$, and a domain monitor machine specification, $\mathbb{M}_{\text{MO}}$, one can develop requirements, $\mathbb{R}_{\text{MC}}$, for, and from such a $\mathbb{R}_{\text{MC}}$ one can develop a domain monitor & controller machine specification $\mathbb{M}_{\text{MC}}$ such that $(\mathbb{D}; \mathbb{M}_{\text{MO}}), \mathbb{M}_{\text{MC}} \models \mathbb{R}_{\text{MC}}$.

3.5.2 Chains of Verifiable Developments

The above illustrated just one chain (A–E) of developments. There are others. All are shown in Fig. 3.



Legend: $\mathbb{D}$ domain, $\mathbb{R}$ requirements, $\mathbb{M}$ machine
DE:DEMO, SI: SIMULATOR, MO: MONITOR, MC: MONITOR & CONTROLLER

Figure 3: Chains of Verifiable Developments

Figure 3 can also be interpreted as prescribing a widest possible range of machine cum software products [Bos00, PBvdL05] for a given domain. One domain may give rise to many different kinds of DEmo machines, SImulators, MOnitors and Monitor & Controllers (the unprimed versions of the $\mathbb{M}_{\text{T}}$ machines (where T ranges over DE, SI, MO, MC)). For each of these there are similarly, "exponentially" many variants of successor machines (the primed versions of the $\mathbb{M}_{\text{T}}$ machines). What does it mean that a machine is a primed version? Well, here it means, for example, that $\mathbb{M}'_{\text{SI}}$ embodies facets of the demo machine $\mathbb{M}_{\text{DE}}$, and that $\mathbb{M}'''_{\text{MC}}$ embodies facets of the demo machine $\mathbb{M}_{\text{DE}}$, of the simulator $\mathbb{M}'_{\text{SI}}$, and the monitor $\mathbb{M}''_{\text{MO}}$. Whether such requirements are desirable is left to product customers and their software providers [Bos00, PBvdL05] to decide.

4 Conclusion

Our divertimento is almost over. It is time to conclude.

4.1 Discussion

The $\mathbb{D}, \mathbb{M} \models \mathbb{R}$ ('correctness' of) development relation appears to have been first indicated in the Computational Logic Inc. `Stack` [BJMY89, GY91] and the EU ESPRIT `ProCoS` [Bjø89, Bjø92] projects; [GGJZ00] presents this same idea with a purpose much like ours, but with more technical discussions.

The term 'domain engineering' appears to have at least two meanings: the one used here [Bjø07, Bjø10a] and one [Har02, FGD02, BHS07] emerging out of the Software Engineering Institute at CMU where it is also called *product line engineering*[5]. Our meaning, is, in a sense, more narrow, but then it seems to also be more highly specialised (with detailed description and formalisation principles and techniques). Fig. 3 on the preceding page illustrates, in capsule form, what we think is the CMU/SEI meaning. The relationship between, say Fig. 3 and *model-based software development* seems obvious but need be explored. An extensive discussion of the term 'domain', as it appears in the software engineering literature is found in [Bjø16b, Sect. 5.3].

4.1.1 What Have We Achieved

We have characterised a spectrum of strongly domain-related as well as strongly inter-related (cf. Fig. 3) software product families: *demos*, *simulators*, *monitors* and *monitor & controllers*. We have indicated varieties of these: simulators based on demos, monitors based on simulators, monitor & controllers based on monitors, in fact any of the latter ones in the software product family list as based on any of the earlier ones. We have sketched temporal relations between simulation traces and domain behaviours: *a priori*, *a posteriori*, *macroscopic* and *microscopic*, and we have identified the real-time cases which lead on to monitors and monitor & controllers.

4.1.2 What Have We Not Achieved — Some Conjectures

We have not characterised the software product family relations other than by the $\mathbb{D}, \mathbb{M} \models \mathbb{R}$ and $(\mathbb{D}; \mathbb{M}_{\mathrm{XYZ}}), \mathbb{M} \models \mathbb{R}$ clauses. That is, we should like to prove conjectured type theoretic inclusion relations like:

$$\wp([\![\mathcal{M}_{\mathrm{X}_{\underline{\mathsf{mod}}\ ext.}}]\!]) \supseteq \wp([\![\mathcal{M}'^{\ldots'}_{\mathrm{X}_{\underline{\mathsf{mod}}\ ext.}}]\!]), \qquad \wp([\![\mathcal{M}'^{\ldots'}_{\mathrm{X}_{\underline{\mathsf{mod}}\ ext.}}]\!]) \supseteq \wp([\![\mathcal{M}''^{\ldots.'}_{\mathrm{X}_{\underline{\mathsf{mod}}\ ext.}}]\!])$$

where X and Y range appropriately, where $[\![\mathcal{M}]\!]$ expresses the meaning of $\mathcal{M}$, where $\wp([\![\mathcal{M}]\!])$ denote the space of all machine meanings and where $\wp([\![\mathcal{M}_{x_{\underline{\mathsf{mod}}\ ext.}}]\!])$ is intended to denote that space modulo ("free of") the y facet (here *ext.*, for extension).

That is, it is conjectured that the set of more specialised, i.e., n primed, machines of kind x is type theoretically "contained" in the set of m primed (unprimed) x machines ($0 \leq m < n$).

There are undoubtedly many such interesting relations between the DEMO, SIMULATOR, MONITOR and MONITOR & CONTROLLER machines, unprimed and primed.

4.1.3 What Should We Do Next

This paper has the subtitle: *A Divertimento of Ideas and Suggestions*. It is not a proper theoretical paper. It tries to throw some light on families and varieties of software, i.e., their relations. It focuses, in particular, on so-called DEMO, SIMULATOR, MONITOR and MONITOR & CONTROLLER software and

[5] http://en.wikipedia.org/wiki/Domain_engineering.

their relation to the "originating" domain, i.e., that in which such software is to serve, and hence that which is being *extended* by such software, cf. the compounded 'domain' $(\mathbb{D};\mathbb{M}_i)$ of in $(\mathbb{D};\mathbb{M}_i),\mathbb{M}_j \models \mathbb{D}$. These notions should be studied formally. All of these notions: requirements projection, instantiation, determination and extension can be formalised; and the specification language, in the form used here (without CSP processes, [Hoa85]) has a formal semantics and a proof system — so the various notions of development, $(\mathbb{D};\mathbb{M}_i),\mathbb{M}_j \models \mathbb{R}$ and $\wp(\mathbb{M})$ can be formalised.

5 Bibliography

References

[BE10] Dines Bjørner and Asger Eir. Compositionality: Ontology and Mereology of Domains. Some Clarifying Observations in the Context of Software Engineering in July 2008, eds. Martin Steffen, Dennis Dams and Ulrich Hannemann. In *Festschrift for Prof. Willem Paul de Roever Concurrency, Compositionality, and Correctness*, volume 5930 of *Lecture Notes in Computer Science*, pages 22–59, Heidelberg, July 2010. Springer.

[BHS07] F. Buschmann, K. Henney, and D.C. Schmidt. *Pattern-Oriented Software Architecture: On Patterns and Pattern Languages*. John Wiley & Sons Ltd., England, 2007.

[BJMY89] W.R. Bevier, W.A. Hunt Jr., J Strother Moore, and W.D. Young. An approach to system verification. *Journal of Automated Reasoning*, 5(4):411–428, December 1989. Special Issue on System Verification.

[Bjø89] Dines Bjørner. A ProCoS Project Description. *Published in two slightly different versions: (1) EATCS Bulletin, October 1989, (2) (Ed. Ivan Plander:) Proceedings: Intl. Conf. on AI & Robotics, Strebske Pleso, Slovakia, Nov. 5-9, 1989, North-Holland, Publ.*, Dept. of Computer Science, Technical University of Denmark, October 1989.

[Bjø92] Dines Bjørner. Trustworthy Computing Systems: The ProCoS Experience. In *14'th ICSE: Intl. Conf. on Software Eng., Melbourne, Australia*, pages 15–34. ACM Press, May 11–15 1992.

[Bjø00] Dines Bjørner. Formal Software Techniques in Railway Systems. In Eckehard Schnieder, editor, *9th IFAC Symposium on Control in Transportation Systems*, pages 1–12, Technical University, Braunschweig, Germany, 13–15 June 2000. VDI/VDE-Gesellschaft Mess– und Automatisieringstechnik, VDI-Gesellschaft für Fahrzeug– und Verkehrstechnik. Invited talk.

[Bjø02] Dines Bjørner. Domain Models of "The Market" — in Preparation for E–Transaction Systems. In *Practical Foundations of Business and System Specifications (Eds.: Haim Kilov and Ken Baclawski)*, The Netherlands, December 2002. Kluwer Academic Press. **URL: http://www2.imm.dtu.dk/~dibj/themarket.pdf**.

[Bjø07] Dines Bjørner. Domain Theory: Practice and Theories, Discussion of Possible Research Topics. In *ICTAC'2007*, volume 4701 of *Lecture Notes in Computer Science (eds. J.C.P. Woodcock et al.)*, pages 1–17, Heidelberg, September 2007. Springer.

[Bjø08] Dines Bjørner. From Domains to Requirements. In *Montanari Festschrift*, volume 5065 of *Lecture Notes in Computer Science (eds. Pierpaolo Degano, Rocco De Nicola and José Meseguer)*, pages 1–30, Heidelberg, May 2008. Springer. **URL: http://www.imm.dtu.dk/~dibj/montanari.pdf**.

[Bjø09a] Dines Bjørner. An Emerging Domain Science – A Rôle for Stanisław Leśniewski's Mereology and Bertrand Russell's Philosophy of Logical Atomism. *Higher-order and Symbolic Computation*, 2009.

[Bjø09b] Dines Bjørner. *Domain Engineering: Technology Management, Research and Engineering*. Research Monograph (# 4); JAIST Press, 1-1, Asahidai, Nomi, Ishikawa 923-1292 Japan, This Research Monograph contains the following main chapters:

1. *On Domains and On Domain Engineering – Prerequisites for Trustworthy Software – A Necessity for Believable Management*, pages 3–38.
2. *Possible Collaborative Domain Projects – A Management Brief*, pages 39–56.
3. *The Rôle of Domain Engineering in Software Development*, pages 57–72.
4. *Verified Software for Ubiquitous Computing – A VSTTE Ubiquitous Computing Project Proposal*, pages 73–106.
5. *The Triptych Process Model – Process Assessment and Improvement*, pages 107–138.
6. *Domains and Problem Frames – The Triptych Dogma and M.A.Jackson's PF Paradigm*, pages 139–175.
7. *Documents – A Rough Sketch Domain Analysis*, pages 179–200.
8. *Public Government – A Rough Sketch Domain Analysis*, pages 201–222.
9. *Towards a Model of IT Security — – The ISO Information Security Code of Practice – An Incomplete Rough Sketch Analysis*, pages 223–282.
10. *Towards a Family of Script Languages – – Licenses and Contracts – An Incomplete Sketch*, pages 283–328.

2009.

[Bjø09c] Dines Bjørner. On Mereologies in Computing Science. In *Festschrift: Reflections on the Work of C.A.R. Hoare*, History of Computing (eds. Cliff B. Jones, A.W. Roscoe and Kenneth R. Wood), pages 47–70, London, UK, 2009. Springer. **URL: http://www2.imm.dtu.dk/~dibj/bjorner-hoare75-p.pdf**.

[Bjø10a] Dines Bjørner. Domain Engineering. In Paul Boca and Jonathan Bowen, editors, *Formal Methods: State of the Art and New Directions*, Eds. Paul Boca and Jonathan Bowen, pages 1–42, London, UK, 2010. Springer.

[Bjø10b] Dines Bjørner. Domain Science & Engineering – *From Computer Science to The Sciences of Informatics, Part I of II: The Engineering Part*. *Kibernetika i sistemny analiz*, (4):100–116, May 2010.

[Bjø11a] Dines Bjørner. Believable Software Management. *Encyclopedia of Software Engineering*, 1(1):1–32, 2011.

[Bjø11b] Dines Bjørner. Domain Science & Engineering – *From Computer Science to The Sciences of Informatics Part II of II: The Science Part*. *Kibernetika i sistemny analiz*, (2):100–120, May 2011.

[Bjø16a] Dines Bjørner. From Domain Descriptions to Requirements Prescriptions – A Different Approach to Requirements Engineering. Technical report, Technical University of Denmark, Fredsvej 11, DK-2840 Holte, Denmark, 2016. Extensive revision of [Bjø08] **URL: http://www2.compute.dtu.dk/~dibj/2015/faoc-req/faoc-req.pdf**.

[Bjø16b] Dines Bjørner. Manifest Domains: Analysis & Description. *Formal Aspects of Computing*, 29(2):175–225, Online: July 2016. **URL: https://doi.org/10.1007/s00165-016-0385-z (doi: 10.1007/s00165-016-0385-z)**.

[Bjø18] Dines Bjørner. Domain Facets: Analysis & Description. Technical report, Technical University of Denmark, Fredsvej 11, DK-2840 Holte, Denmark, May 2018. Extensive revision of [Bjø10a]. **URL: http://www.imm.dtu.dk/~dibj/2016/facets/faoc-facets.pdf**.

[Bos00] Jan Bosch. *Design and Use of Software Architectures: Adopting and Evolving a Product-line Approach*. ACM Press/Addison-Wesley, New York, NY, 2000.

[FGD02] R. Falbo, G. Guizzardi, and K.C. Duarte. An Ontological Approach to Domain Engineering. In *Software Engineering and Knowledge Engineering*, Proceedings of the 14th international conference SEKE'02, pages 351–358, Ischia, Italy, July 15-19 2002. ACM.

[GGJZ00] Carl A. Gunter, Elsa L. Gunter, Michael A. Jackson, and Pamela Zave. A Reference Model for Requirements and Specifications. *IEEE Software*, 17(3):37–43, May–June 2000.

[GY91] Don I. Good and William D. Young. Mathematical Methods for Digital Systems Development. In *VDM '91: Formal Software Development Methods*, pages 406–430. Springer-Verlag, October 1991. Volume 2.

[Har02] M. Harsu. A Survey on Domain Engineering. Review, Institute of Software Systems, Tampere University of Technology, Finland, December 2002.

[Hoa85] C.A.R. Hoare. *Communicating Sequential Processes*. C.A.R. Hoare Series in Computer Science. Prentice-Hall International, 1985. Published electronically: http://www.usingcsp.com/cspbook.pdf (2004).

[PBvdL05] K. Pohl, G. Bockle, and F. van der Linden. *Software Product Line Engineering*. Springer, Berlin, Heidelberg, New York, 2005.

A A Domain Description Example: A Credit Card System

This appendix section presents a first attempt at a model of a credit card system.

We present a domain description of an abstracted credit card system. The narrative part of the description is terse, perhaps a bit too terse.

Credit cards are moving from simple plastic cards to smart phones. Uses of credit cards move from their mechanical insertion in credit card terminals to being swiped. Authentication (hence not modeled) moves from keying in security codes to eye iris "prints", and/or finger prints and/or voice prints or combinations thereof.

The description of this section abstracts from all that in order to understand a bare, minimum essence of credit cards and their uses. Based on a model, such as presented here, the reader should be able to extend/refine the model into any future technology – for requirements purposes.

A.1 Endurants

A.1.1 Credit Card Systems

Sect. **??** *on page ??: observe_part_sorts*

1. Credit card systems, $ccs{:}CCS$, consists of three kinds of parts:
2. an assembly, $cs{:}CS$, of credit cards[6],
3. an assembly, $bs{:}BS$, of banks, and
4. an assembly, $ss{:}SS$, of shops.

```
type
1   CCS
2   CS
3   BS
4   SS
value
2   obs_part_CS: CCS → CS
3   obs_part_BS: CCS → BS
4   obs_part_SS: CCS → SS
```

[6] We "equate" credit cards with their holders.

The composite part CS can be thought of as a credit card company, say VISA[7]. The composite part BS can be thought of as a bank society, say BBA: British Banking Association. The composite part SS can be thought of as the association of retailers, say bira: British Independent Retailers Association[8].

Sect. **??** *on page ??: observe_part_type*

5. There are credit cards, $c{:}C$, banks $b{:}B$, and shops $s{:}S$.
6. The credit card part, $cs{:}CS$, abstracts a set, $soc{:}Cs$, of card.
7. The bank part, $bs{:}BS$, abstracts a set, $sob{:}Bs$, of banks.
8. The shop part, $ss{:}SS$, abstracts a set, $sos{:}Ss$, of shops.

```
type
5  C, B, S
6  Cs = C-set
7  Bs = B-set
8  Ss = S-set
value
6  obs_part_CS: CS → Cs, obs_part_Cs: CS → Cs
7  obs_part_BS: BS → Bs, obs_part_Bs: BS → Bs
8  obs_part_SS: SS → Ss, obs_part_Ss: SS → Ss
```

Sect. **??** *on page ??: observe_unique_identifier*

9. Assembliers of credit cards, banks and shops have unique identifiers, $csi{:}CSI$, $bsi{:}BSI$, and $ssi{:}SSI$.
10. Credit cards, banks and shops have unique identifiers, $ci{:}CI$, $bi{:}BI$, and $si{:}SI$.
11. One can define functions which extract all the
12. unique credit card,
13. bank and
14. shop identifiers from a credit card system.

```
9   CSI, BSI, SSI
10   CI, BI, SI
value
9  uid_CS: CS→CSI, uid_BS: BS→BSI, uid_SS: SS→SSI,
10  uid_C: C→CI, uid_B: B→BI, uid_S: S→SI,
12  xtr_CIs: CCS → CI-set
12  xtr_CIs(ccs) ≡ {uid_C(c)|c:C•c ∈ obs_part_Cs(obs_part_CS(ccs))}
13  xtr_BIs: CCS → BI-set
13  xtr_BIs(ccs) ≡ {uid_B(s)|b:B•b ∈ obs_part_Bs(obs_part_BS(ccs))}
14  xtr_SIs: CCS → SI-set
14  xtr_SIs(ccs) ≡ {uid_S(s)|s:S•s ∈ obs_part_Ss(obs_part_SS(ccs))}
```

[7] Our simple model allows for only one credit card company. But that model can easily be extended to model a set of credit card companies, viz.: VISA, MasterCard, American Express, Diner's Club, etc..

[8] The model does not prevent "shops" from being airlines, or car rental agencies, or dentists, or consultancy firms. In this case SS would be some appropriate association.

15. For all credit card systems it is the case that
16. all credit card identifiers are distinct from bank identifiers,
17. all credit card identifiers are distinct from shop identifiers,
18. all shop identifiers are distinct from bank identifiers,

```
axiom
15  ∀ ccs:CCS •
15    let cis=xtr_CIs(ccs), bis=xtr_BIs(ccs), sis = xtr_SIs(ccs) in
16    cis ∩ bis = {}
17  ∧ cis ∩ sis = {}
18  ∧ sis ∩ bis = {} end
```

A.1.2 Credit Cards

Sect. ?? on page ??: observe_mereology and *Sect. ?? on page ??: observe_part_attributes*

19. A credit card has a mereology which "connects" it to any of the shops of the system and to exactly one bank of the system,
20. and some attributes — which we shall presently disregard.
21. The wellformedness of a credit card system includes the wellformedness of credit card mereologies with respect to the system of banks and shops:
22. The unique shop identifiers of a credit card mereology must be those of the shops of the credit card system; and
23. the unique bank identifier of a credit card mereology must be of one of the banks of the credit card system.

```
type
19.    CM = SI-set × BI
value
19.    obs_mereo_CM: C → CM
21     wf_CM_of_C: CCS → Bool
21     wf_CM_of_C(ccs) ≡
19       let bis=xtr_BIs(ccs), sis=xtr_SIs(ccs) in
19         ∀ c:C•c ∈ obs_part_Cs(obs_part_CS(ccs)) ⇒
19           let (ccsis,bi)=obs_mereo_CM(c) in
22           ccsis ⊆ sis
23         ∧ bi ∈ bis
19       end end
```

Constraint 22 limits a credit card to potentially be used only in a proper subset of all shops. To allow for all shops one must change the wording to *'be* **all** *those of the shops ...'*, and change ⊆ in formula line 22 to '='.

A.1.3 Banks

Sect. ?? on page ??: observe_mereology and *Sect. ?? on page ??: observe_part_attributes*

Our model of banks is (also) very limited.

24. A bank has a mereology which "connects" it to a subset of all credit cards and a subset of all shops,
25. and, as attributes:
26. a cash register, and
27. a ledger.
28. The ledger records for every card, by unique credit card identifier,
29. the current balance: how much money, credit or debit, i.e., plus or minus, that customer is owed, respectively has borrowed from the bank,
30. the dates-of-issue and -expiry of the credit card, and
31. the name, address, and other information about the credit card holder.
32. The wellformedness of the credit card system includes the wellformedness of the banks with respect to the credit cards and shops:
33. the bank mereology's
34. must list a subset of the credit card identifiers and a subset of the shop identifiers.

```
type
24   BM = CI-set × SI-set
26   CR = Bal
27   LG = CI ~m~> (Bal×DoI×DoE×...)
29   Bal = Int
value
24   obs_mereo_B: B → BM
26   attr_CR: B → CR
27   attr_LG: B → LG
32       wf_BM_B: CCS → Bool
32       wf_BM_B(ccs) ≡
32           let allcis = xtr_CIs(ccs), allsis = xtr_SIs(ccs) in
32           ∀ b:B • b ∈ obs_part_Bs(obs_part_BS(ccs)) in
33               let (cis,sis) = obs_mereo_B(b) in
34               cis ⊆ ∀ cis ∧ sis ⊆ allsis
32           end end
```

A.1.4 Shops

35. The mereology of a shop is a pair: a unique bank identifiers, and a set of unique credit card identifiers.
36. The mereology of a shop

37. must list a bank of the credit card system,
38. band a subset (or all) of the unique credit identifiers.

We omit treatment of shop attributes.

```
type
35  SM = CI-set × BI
value
35  obs_mereo_S: S → SM
36  wf_SM_S: CCS → Bool
36  wf_SM_S(ccs) ≡
36    let allcis = xtr_CIs(ccs), allbis = xtr_BIs(ccs) in
36    ∀ s:S • s ∈ obs_part_Ss(obs_part_SS(ccs)) ⇒
36        let (cis,bi) obs_mereo_S(s) in
37        bi ∈ allbis
38      ∧ cis ⊆ allcis
36    end end
```

A.2 Perdurants

A.2.1 Behaviours

[Bjø16b, Sect.4.11.2, pg.36]: Process Schema I: Abstract `is_composite(p)`
[Bjø16b, Sect.4.11.2, pg.37]: Process Schema II: Concrete `is_concrete(p)`

39. We ignore the behaviours related to the CCS, CS, BS and SS parts.
40. We therefore only consider the behaviours related to the Cs, Bs and Ss parts.
41. And we therefore compile the credit card system into the parallel composition of the parallel compositions of all the credit card, crd, all the bank, bnk, and all the shop, shp, behaviours.

```
value
39  ccs:CCS
39  cs:CS = obs_part_CS(ccs),
39  uics:CSI =uid_CS(cs),
39  bs:BS = obs_part_BS(ccs),
39  uibs:BSI =uid_BS(bs),
39  ss:SS = obs_part_SS(ccs),
39  uiss:SSI =uid_SS(ss),
40  socs:Cs = obs_part_Cs(cs),
40  sobs:Bs = obs_part_Bs(bs),
40  soss:Ss = obs_part_Ss(ss),

value
41  sys: Unit → Unit,
39  sys() ≡
41        cards_uics(obs_mereo_CS(cs),...) ‖ ‖ {crd_uid_C(c)(obs_mereo_C(c))|c:C•c ∈ socs}
41    ‖ banks_uibs(obs_mereo_BS(bs),...) ‖ ‖ {bnk_uid_B(b)(obs_mereo_B(b))|b:B•b ∈ sobs}
41    ‖ shops_uiss(obs_mereo_SS(ss),...) ‖ ‖ {shp_uid_S(s)(obs_mereo_S(s))|s:S•s ∈ soss},
39  cards_uics(...) ≡ skip,
39  banks_uibs(...) ≡ skip,
39  shops_uiss(...) ≡ skip

axiom  skip ‖ behaviour(...) ≡ behaviour(...)
```

A.2.2 Channels

[Bjø16b, Sect. 4.5.1, pg.31]: Channels and Communications
[Bjø16b, Sect. 4.5.2, pg.31]: Relations Between Attributes Sharing and Channels

42. Credit card behaviours interact with bank (each with one) and many shop behaviours.
43. Shop behaviours interact with bank (each with one) and many credit card behaviours.
44. Bank behaviours interact with many credit card and many shop behaviours.

    The inter-behaviour interactions concern:

45. between credit cards and banks: withdrawal requests as to a sufficient, mk_Wdr(am), balance on the credit card account for buying am:AM amounts of goods or services, with the bank response of either is_OK() or is_NOK(), or the revoke of a card;
46. between credit cards and shops: the buying, for an amount, am:AM, of goods or services: mk_Buy(am), or the refund of an amount;
47. between shops and banks: the deposit of an amount, am:AM, in the shops' bank account: mk_Depost(ui,am) or the removal of an amount, am:AM, from the shops' bank account: mk_Removl(bi,si,am)

channel
42 {ch_cb[ci,bi]|ci:CI,bi:BI•ci ∈ cis ∧ bi ∈ bis}:CB_Msg
43 {ch_cs[ci,si]|ci:CI,si:SI•ci ∈ cis ∧ si ∈ sis}:CS_Msg
44 {ch_sb[si,bi]|si:SI,bi:BI•si ∈ sis ∧ bi ∈ bis}:SB_Msg
45 CB_Msg == mk_Wdrw(am:aM) | is_OK() | is_NOK() | ...
46 CS_Msg == mk_Buy(am:aM) | mk_Ref(am:aM) | ...
47 SB_Msg == Depost | Removl | ...
47 Depost == mk_Dep((ci:CI|si:SI),am:aM) |
47 Removl == mk_Rem(bi:BI,si:SI,am:aM)

A.2.3 Behaviour Interactions

48. The credit card initiates
    a) buy transactions
        i. [1.Buy] by enquiring with its bank as to sufficient purchase funds (am:aM);
        ii. [2.Buy] if NOK then there are presently no further actions; if OK
        iii. [3.Buy] the credit card requests the purchase from the shop – handing it an appropriate amount;
        iv. [4.Buy] finally the shop requests its bank to deposit the purchase amount into its bank account.
    b) refund transactions
        i. [1.Refund] by requesting such refunds, in the amount of am:aM, from a[ny] shop; whereupon
        ii. [2.Refund] the shop requests its bank to move the amount am:aM from the shop's bank account
        iii. [3.Refund] to the credit card's account.

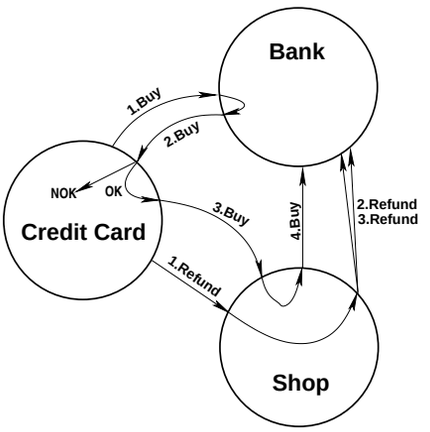


Figure 4: Credit Card, Bank and Shop Behaviours

Thus the three sets of behaviours, crd, bnk and shp interact as sketched in Fig. 4.

[1.Buy]	Item 54, Pg.21	card	ch_cb[ci,bi]!mk_Wdrw(am) (shown as ... three lines down) and
	Item 63, Pg.22	bank	mk_Wdrw(ci,am)=[]{ch_cb[bi,bi]?|ci:CI•ci ∈ cis}.
[2.Buy]	Items 56-57, Pg.21	bank	ch_cb[ci,bi]!is_[N]OK() and
	Item 54, Pg.21	shop	(...;ch_cb[ci,bi]?).
[3.Buy]	Item 56, Pg.21	card	ch_cs[ci,si]!mk_Buy(am) and
	Item 78, Pg.23	shop	mk_Buy(am)=[]{ch_cs[ci,si]?|ci:CI•ci∈ cis}.
[4.Buy]	Item 79, Pg.23	shop	ch_sb[si,bi]!mk_Dep(si,am) and
	Item 68, Pg.22	bank	mk_Dep(si,am)=[]{ch_cs[ci,si]?|si:SI•si∈sis}.
[1.Refund]	Item 60, Pg.21	card	ch_cs[ci,si]!mk_Ref((ci,si),am) and
	Item 79, Pg.23	shop	(si,mk_Ref(ci,am))=[]{si′,ch_sb[si,bi]?|si,si′:SI•{si,si′}⊆sis∧si=si′}.
[2.Refund]	Item 83, Pg.24	shop	ch_sb[si,cbi]!mk_Ref(cbi,(ci,si),am and
	Item 72, Pg.23	bank	(si,mk_Ref(cbi,(ci,am)))=[]{(si′,ch_sb[si,bi]?)|si,si′:SI•{si,si′}⊆sis∧si=si′}.
[3.Refund]	Item 84, Pg.24	shop	ch_sb[si,sbi]!mk_Wdr(si,am)) **end** and
	Item 73, Pg.23	bank	(si,mk_Wdr(ci,am))=[]{(si′,ch_sb[si,bi]?)|si,si′:SI•{si,si′}⊆sis∧si=si′}

A.2.4 Credit Card

[Bjø16b, Sect. 4.11.2, pg. 37]: Processs Schema III: `is_atomic(p)`

49. The credit card behaviour, crd, takes the credit card unique identifier, the credit card mereology, and attribute arguments (omitted). The credit card behaviour, crd, accepts inputs from and offers outputs to the bank, bi, and any of the shops, si∈sis.
50. The credit card behaviour, crd, non-deterministically, internally "cycles" between buying and getting refunds.

value
49 $\text{crd}_{ci:CI}$: (bi,sis):CM → **in**,**out** ch_cb[ci,bi],{ch_cs[ci,si]|si:SI•si ∈ sis} **Unit**
49 crd_{ci}(bi,sis) ≡ (buy(ci,(bi,sis)) ⌈⌉ ref(ci,(bi,sis))) ; crd_{ci}(ci,(bi,sis))

[Bjø16b, Sect. 4.11.2, pg. 38]: Process Schemas IV–V: Core Processes (I–II)

51. By am:AM we mean an amount of money, and by si:SI we refer to a shop in which we have selected a number or goods or services (not detailed) costing am:AM.
52. The buyer action is simple.
53. The amount for which to buy and the shop from which to buy are selected (arbitrarily).
54. The credit card (holder) withdraws am:AM from the bank, if sufficient funds are available[9].
55. The response from the bank
56. is either OK and the credit card [holder] completes the purchase by buying the goods or services offered by the selected shop,
57. or the response is "not OK", and the transaction is skipped.

```
type
51  AM = Int
value
52  buy: ci:CI × (bi,sis):CM → in,out ch_cb[ ci,bi ] out {ch_cs[ ci,si ]|si:SI•si ∈ sis}  Unit
52  buy(ci,(bi,sis)) ≡
53      let am:aM • am>0, si:SI • si ∈ sis in See Discussion note 87a on page 24
54      let msg = (ch_cb[ ci,bi ]!mk_Wdrw(am);ch_cb[ ci,bi ]?) in
55      case msg of
56          is_OK() → ch_cs[ ci,si ]!mk_Buy(am),
57          is_NOK() → skip
52      end end end
```

58. The refund action is simple.
59. The credit card [handler] requests a refund am:AM
60. from shop si:SI.

    This request is handled by the shop behaviour's sub-action *ref*, see lines 76.–85. page 24.

```
value
58  rfu: ci:CI × (bi,sis):CM → out {ch_cs[ ci,si ]|si:SI•si ∈ sis}  Unit
58  rfu(ci,(bi,sis)) ≡
59      let am:AM • am>0, si:SI • si ∈ sis in   See Discussion note 87b on page 24
60      ch_cs[ ci,si ]!mk_Ref(bi,(ci,si),am)
58      end
```

[9] First the credit card [holder] requests a withdrawal. If sufficient funds are available, then the withdrawal takes place, otherwise not – and the credit card holder is informed accordingly.

A.2.5 Banks

[Bjø16b, Sect. 4.11.2, pg. 37]: Processs Schema III: `is_atomic(p)`

61. The bank behaviour, bnk, takes the bank's unique identifier, the bank mereology, and the programmable attribute arguments: the ledger and the cash register. The bank behaviour, bnk, accepts inputs from and offers outputs to the any of the credit cards, $ci \in cis$, and any of the shops, $si \in sis$.

62. The bank behaviour non-deterministically externally chooses to accept either 'withdraw'al requests from credit cards or 'deposit' requests from shops or 'refund' requests from credit cards.

```
value
61  bnk_bi:BI: (cis,sis):BM → (LG×CR) →
61      in,out {ch_cb[ ci,bi ]|ci:CI•ci ∈ cis} {ch_sb[ si,bi ]|si:SI•si ∈ sis}  Unit
61  bnk_bi((cis,sis))(lg:(bal,doi,doe,...),cr) ≡
62      wdrw(bi,(cis,sis))(lg,cr) [] depo(bi,(cis,sis))(lg,cr) [] refu(bi,(cis,sis))(lg,cr)
```

63. The 'withdraw' request, wdrw, (an action) non-deterministically, externally offers to accept input from a credit card behaviour and marks the only possible form of input from credit cards, mk_Wdrw(ci,am), with the identity of the credit card.

64. If the requested amount (to be withdrawn) is not within balance on the account

65. then we, at present, refrain from defining an outcome (**chaos**), whereupon the bank behaviour is resumed with no changes to the ledger and cash register;

66. otherwise the bank behaviour informs the credit card behaviour that the amount can be withdrawn; whereupon the bank behaviour is resumed notifying a lower balance and 'withdraws' the monies from the cash register.

```
value
62  wdrw: bi:BI × (cis,sis):BM → (LG×CR) → in,out {ch_cb[ bi,ci ]|ci:CI•ci ∈ cis}  Unit
62  wdrw(bi,(cis,sis))(lg,cr) ≡
63      let mk_Wdrw(ci,am) = [] {ch_cb[ ci,bi ]?|ci:CI•ci ∈ cis} in
62      let (bal,doi,doe) = lg(ci) in
64      if am>bal
65          then (ch_cb[ ci,bi ]!is_NOK(); bnk_bi(cis,sis)(lg,cr))
66          else (ch_cb[ ci,bi ]!is_OK(); bnk_bi(cis,sis)(lg†[ ci↦(bal−am,doi,doe) ],cr−am)) end
61      end end
```

The ledger and cash register attributes, lg,cr, are programmable attributes. Hence they are modeled as separate function arguments, cf. Sect. 4.7.3, Page 33 in [Bjø16b].

67. The deposit action is invoked, either by a shop behaviour, when a credit card [holder] buy's for a certain amount, am:AM, or requests a refund of that amount. The deposit is made by shop behaviours, either on behalf of themselves, hence am:AM, is to be inserted into the shops' bank account, si:SI, or on behalf of a credit card [i.e., a customer], hence am:AM, is to be inserted into the credit card holder's bank account, si:SI.

68. The message, ch_cs[ci,si]?, received from a credit card behaviour is either concerning a buy [in which case i is a ci:CI, hence sale, or a refund order [in which case i is a si:SI].

69. In either case, the respective bank account is "upped" by am:AM – and the bank behaviour is resumed.

```
value
67  deposit: bi:BI × (cis,sis):BM → (LG×CR) → in,out {ch_sb[ bi,si ]|si:SI•si ∈ sis} Unit
67  deposit(bi,(cis,sis))(lg,cr) ≡
68      let mk_Dep(si,am) = [] {ch_cs[ ci,si ]?|si:SI•si ∈ sis} in
67      let (bal,doi,doe) = lg(si) in
69      bnk_bi(cis,sis)(lg†[ si↦(bal+am,doi,doe) ],cr+am)
67      end end
```

70. The refund action
71. non-deterministically externally offers to either
72. non-deterministically externally accept a mk_Ref(ci,am) request from a shop behaviour, si, or
73. non-deterministically externally accept a mk_Wdr(ci,am) request from a shop behaviour, si.

    The bank behaviour is then resumed with the

74. credit card's bank balance and cash register incremented by am and the
75. shop' bank balance and cash register decremented by that same amount.

```
value
70  rfu: bi:BI × (cis,sis):BM → (LG×CR) → in,out {ch_sb[ bi,si ]|si:SI•si ∈ sis} Unit
70  rfu(bi,(cis,sis))(lg,cr) ≡
72      (let (si,mk_Ref(cbi,(ci,am))) = [] {(si′,ch_sb[ si,bi ]?)|si,si′:SI•{si,si′}⊆sis∧si=si′} in
70      let (balc,doic,doec) = lg(ci) in
74      bnk_bi(cis,sis)(lg†[ ci↦(balc+am,doic,doec) ],cr+am)
70      end end)
71    []
73    (let (si,mk_Wdr(ci,am)) = [] {(si′,ch_sb[ si,bi ]?)|si,si′:SI•{si,si′}⊆sis∧si=si′} in
70      let (bals,dois,does) = lg(si) in
75      bnk_bi(cis,sis)(lg†[ si↦(bals−am,dois,does) ],cr−am)
70      end end)
```

A.2.6 Shops

[Bjø16b, Sect. 4.11.2, pg. 37]: Processs Schema III: `is_atomic(p)`

76. The shop behaviour, shp, takes the shop's unique identifier, the shop mereology, etcetera.
77. The shop behaviour non-deterministically, externally

    either

78. offers to accept a Buy request from a credit card behaviour,
79. and instructs the shop's bank to deposit the purchase amount.

80. whereupon the shop behaviour resumes being a shop behaviour;
81. or
82. offers to accept a refund request in this amount, am, from a credit card [holder].
83. It then proceeds to inform the shop's bank to withdraw the refund from its ledger and cash register,
84. and the credit card's bank to deposit the refund into its ledger and cash register.
85. Whereupon the shop behaviour resumes being a shop behaviour.

```
value
76  shp_si:SI: (CI-set×BI)×...→in,out: {ch_cs[ ci,si ]|ci:CI•ci ∈ cis},{ch_sb[ si,bi′ ]||bi′:BI•bi′isin bis} Unit
76  shp_si((cis,bi),...) ≡
78       (sal(si,(bi,cis),...)
81       ⌷
82       ref(si,(cis,bi),...)):

76  sal: SI×(CI-set×BI)×...→in,out: {cs[ ci,si ]|ci:CI•ci ∈ cis},sb[ si,bi ] Unit
76  sal(si,(cis,bi),...) ≡
78     let mk_Buy(am) = ⌷{ch_cs[ ci,si ]?|ci:CI•ci ∈ cis} in
79     ch_sb[ si,bi ]!mk_Dep(si,am) end ;
80     shp_si((cis,bi),...)

76  ref: SI×(CI-set×BI)×...→in,out: {ch_cs[ ci,si ]|ci:CI•ci ∈ cis},{ch_sb[ si,bi′ ]||bi′:BI•bi′isin bis} Unit
82  ref(si,(cis,sbi),...) ≡
82     let mk_Ref((ci,cbi,si),am) = ⌷{ch_cs[ ci,si ]?|ci:CI•ci ∈ cis} in
83     (ch_sb[ si,cbi ]!mk_Ref(cbi,(ci,si),am)
84   ‖ ch_sb[ si,sbi ]!mk_Wdr(si,am)) end ;
85     shp_si((cis,sbi),...)
```

A.3 Discussion

86. The credit card system narrated and formalised in this document is an abstraction. We claim that it portrays an essence of credit cards.
87. The reader may object to certain things:
    a) We do not model how a credit card holder selects services from a service provider (here modelled as shops) or products in a shop. Nor do we model that the card holder actually obtains those services or products.
       All this is summarised in Item A.2.4 on page 21: **let** am:aM • am>0, si:SI • si ∈ sis **in**.
       In other words: this is not considered an element of "an essence" of credit cards.
    b) We, "similarly" do not model how the refund request is arrived at.
       All this is summarised in Item A.2.4 on page 21: **let** am:AM • am>0, si:SI • si ∈ sis **in**.
       In other words: this is not considered an element of "an essence" of credit cards.
    c) Also: we do not model whether the balance of the shop's bank account is sufficient to refund a card holder.

d) Etcetera.

The present credit card system model can "easily" be extended to incorporate these and other matters.

88. Without showing explicit evidence we claim that present domain description can serve as a basis for both the domain and requirements modeling of standard as well as current and future credit/pay/etc. card systems.

89. Etcetera.