Computer Languages, Systems & Structures 54 (2018) 297–315

Contents lists available at ScienceDirect

Computer Languages, Systems & Structures

journal homepage: www.elsevier.com/locate/cl

An Event-B-based approach to hybrid systems engineering and its application to a hemodialysis machine case study

Andreea Buga[a], Atif Mashkoor[b], Sorana Tania Nemeş[c], Klaus-Dieter Schewe[d,*], Pornpan Songprasop[d]

[a] Dynatrace Austria GmbH, Linz, Austria
[b] Software Competence Center Hagenberg GmbH, Hagenberg, Austria
[c] Runtastic GmbH, Pasching, Austria
[d] Laboratory for Client-Centric Cloud Computing, Linz, Austria

ARTICLE INFO

Article history:
Received 2 March 2018
Revised 13 July 2018
Accepted 15 July 2018
Available online 21 July 2018

Keywords:
Systems engineering
Hybrid system
Event-B
Conceptual model
Asynchronous behavior
View
Hemodialysis machine

ABSTRACT

Systems engineering concerns the complete process for the development of complex systems comprising hardware, software, facilities and personnel. Such systems are hybrid, as some components are characterized by continuous behavior, whereas the behavior of others is discrete. In this paper we present a concise conceptual model for hybrid systems engineering with semantics grounded in a hybrid extension of Event-B. We show that structural modeling can be based on well-known concepts of the entity-relationship model requiring only some extensions to data types and constraints, while behavioral modeling requires a careful separation of synchronous and asynchronous interaction and high-level means for the integration of continuous functions. On these grounds we address the separation of concerns for continuous and hybrid components. The article uses a sophisticated industrial example of a hemodialysis machine to illustrate the modeling method.

© 2018 Elsevier Ltd. All rights reserved.

1. Introduction

The last decades have seen an increasing ubiquity of software systems that is commonly accompanied by buzzwords such as “cyber-physical systems”, “internet of things”, “systems of systems”, etc. None of these buzzwords has been precisely defined and despite a common core there are subtle differences concerning what is emphasized by each of them. However, they all share the commonality that software systems appear more and more frequently as parts of complex, hybrid systems, which comprise mechanical, electrical, hydraulic, electronic parts as well as software. Consequently, the integrated development of such systems is a main challenge for systems development and thus also for conceptual modeling. Systems Engineering (SE) is supposed to address exactly this challenge.

The state of the art in SE is dominated by SysML, the OMG Systems Modeling Language (OMG SysML™) [36], and support tools such as PTC Integrity Modeler [38], Magic Draw [33], Enterprise Architect [24] and others. As our analysis of SysML, its associated tools and other related work—see the separate Section 2 on state of the art and related work—reveals, SysML is too close to UML and as such inherits known deficiencies such as a lack of formal semantics. In particular,

* Corresponding author.
E-mail addresses: buga.andreea@gmail.com (A. Buga), atif.mashkoor@scch.at (A. Mashkoor), tania.nemes@gmail.com (S.T. Nemeş), kdschewe@acm.org (K.-D. Schewe), pornpansongprasop@gmail.com (P. Songprasop).

https://doi.org/10.1016/j.cl.2018.07.004
1477-8424/© 2018 Elsevier Ltd. All rights reserved.

the decisive continuous aspects in hybrid systems are treated like wall flowers, even more so in the support tools, and a necessary distinction between synchronous and asynchronous behavior is missing. This implies the need for a more rigorous support of hybrid systems engineering, which is the main motivation for our research.

1.1. Our contribution

In this article we do not deal with the formalization of SysML. Instead, we define a more generic conceptual model for SE grounded in well-established methods for structural conceptual modeling and rigorous specification of behavior. In order to successfully master the inherent complexity of hybrid systems, the structuring into components, the capture of relationships between these components, and the elicitation of constraints that have to be respected by behavioral models, is indispensable. This has among others been emphasized already by Bjørner in his monographs on Software Engineering, where he emphasized *domain engineering* as a highly important and indispensable aspect of the development process [15,16]. This aspect is also highlighted in conceptual modeling as well as in the ground modeling approach that is intrinsic part of the development methodology with Abstract State Machines [19].

There are many approaches for the capture of the structural aspects of a domain. In our conceptual model we decided to adopt the higher-order entity-relationship model (HERM) [48], which comes with a well established formal semantics based on sets, a sophisticated theory of structural constraints, subsumption of other methods, in particular UML class diagrams as employed by SysML, and most importantly a long history of many successful applications of very large size. Furthermore, in hybrid systems we have to capture continuous sequences of states, which can be equivalently captured by states that involve continuous functions as values, thus adding continuous functions to the open type system exploited by HERM is a straightforward, but nonetheless extremely powerful extension that will make the method suitable for structural modeling of hybrid systems.

In order to integrate structure and behavior we adopt concepts from extended rigorous methods[1] for hybrid systems specification. Natural candidates are (hybrid) ASMs [12], (hybrid) Event-B [9,10,46] or TLA $^+$ [31]. In principle each of these rigorous methods can be used for our purposes, as continuous functions as values can be easily added. We therefore aim at an approach, where the role of the rigorous method is to serve for the specification of behavior, and differences in expressiveness allow us to choose the method that is most appropriate for a particular application. This includes the use of supporting tools such as the RODIN extension in [46] to capture hybrid dynamic systems and to enable rigorous refinement towards implementation. Therefore, our work in [21] emphasizes the use of concurrent Abstract State Machines, while in this article we show that Event-B can be used in a similar way. This does not include a statement which one should be preferred in general, but independent of the choice it is rather straightforward to show how SysML can be captured by our model.

Concerning the relationship between ASMs and Event-B both are state-based rigorous methods, so the semantics is expressed by runs, i.e. sequences of states. The exact definition of state is different in both methods, but both can be mapped onto each other, as the mathematical underpinning is provided by universal algebras. It is well known that unbounded parallelism is only supported by ASMs and cannot be expressed in Event-B in a behaviorally equivalent way. However, if we dispense with unbounded parallelism, both sequential ASMs and Event-B can express exactly the same type of algorithms. Börger and Leuschel showed how to define translations between both rigorous methods [32]. Theoretically, the result can also be derived from Gurevich's proof of the sequential ASM thesis [27].

With respect to concurrency, however, an extension has been defined for ASMs [18], but (hybrid) Event-B is de facto bound to single machines. In this article we dare to use multiple Event-B machines with shared state variables (both concept that strictly speaking are alien to Event-B), and we adopt the concurrent run semantics from concurrent ASMs for Event-B with multiple machines[2]. Whether the Event-B community will accept this proposal is open. However, without a proper theory of asynchronously interacting machines a method will not be suitable for hybrid systems development. On the other hand, such an extension by multiple Event-B machines or concurrent ASMs gives a precise meaning to the term "systems of systems".

Concerning the structural part of our conceptual model the adoption of HERM allows us to capture part-of relationships, specialization and generalization, complex attributes and sophisticated constraints. In doing so the model will capture sets of complex devices, and values associated with attributes will define a state of the system corresponding to instances of schemata in HERM. However, in order to capture continuous aspects as well, we adopt the view of hybrid dynamic systems [3,37], according to which continuous, piecewise differentiable functions can be used as values. The discrete values then correspond to piecewise constant functions that are continuous using a discrete topology. The presence of continuous functions as values and the fact that systems components have to be considered as being active further requires to add flow constraints to the model, which simply means that at any time a value may flow out of some component into another one without explicit control. This corresponds to the behavior of "pliant" locations in hybrid ASMs [13].

Concerning the behavioral part of our conceptual model we directly adopt the behavioral theory of concurrency [18] by defining agents and rules associated with them. We carry this semantics forward to Event-B with multiple machines. The events manipulate the "pliant", i.e. continuous, and "mode", i.e. discrete locations, and thus require the use of terms referring to solutions of differential equations, in particular initial value problems.

[1] Note that we emphasize only the specifications, whereas the treatment of refinement is reserved for future work.

[2] These extensions to Event-B are handled in more detail in a forthcoming paper by one of the authors [42].

In this article we continue our research in [20,21], where we developed a conceptual model for hybrid dynamic systems based on HERM for structural modeling and constraints, and on concurrent hybrid ASMs and Event-B, respectively, for behavioral modeling. In our previous work we illustrated our model on the landing gear case study [17], and the hemodialysis machine case study [35], respectively. The latter case study will be further explored in this article.

1.2. Outline of the article

Section 2 provides more details of the state of the art in hybrid systems engineering emphasizing SysML and the use of rigorous methods. In Section 3 we present our structural conceptual model, which is based on meromorphic part-of-relationships. We integrate continuous functions as values associated with attributes, which is in strict analogy to the hybrid extensions to Event-B and ASMs, and show how the structural model gives rise to static components of Event-B machines. We assume familiarity with the basic concepts of Event-B. Section 4 is dedicated to constraints, i.e. structural dependencies, flow dependencies and parametric constraints, the latter ones integrating differential equations. We define the different types of constraints in general, and show how they are captured by invariants in Event-B machines. In Section 5 we present our approach to behavioral modeling. While flow dependencies imply synchronous behavior, we support general asynchronous behavior exploiting the behavioral theory of concurrent systems [18]. We formalize the semantics using multiple Event-B machines with common contexts, and discuss the separation of concerns between continuous and discrete components. In Section 6 we model parts of the hemodialysis machine case study [35] using the structural and behavioral concepts of our model. While this illustrates the concepts in principle, the solution to the case study as such will remain incomplete—more details concerning the structural aspects are described in [43]. We conclude with a brief summary and outlook in Section 7.

In order to see examples for the concepts introduced in Sections 3–5, we recommend to read Section 6 in parallel to the development of our model and to map the specifications in the case study to the formal definitions of our model. In order to support this parallel reading we make extensive use of cross referencing between our definitions in Sections 3–5 and illustrating examples in Section 6.

2. Related work

The state of the art in SE is dominated by SysML, the OMG Systems Modeling Language (OMG SysML™) [36], and support tools such as PTC Integrity Modeler [38], Magic Draw [33], Enterprise Architect [24] and others. SysML provides graphical notations for SE on a high level of abstraction. It has been adapted from the commonly known Universal Modeling Language (UML), thus provides several types of diagrams to capture systems requirements. There are three classes of diagrams:

- structure diagrams, which capture
  - package diagrams as in UML,
  - block definition diagrams adapted from UML class diagrams to capture system components, their attributes and operations, and relationships between components,
  - internal block diagrams capturing further dependencies, in particular flow dependencies, between components and their attributes,
  - parametric diagrams, which define detailed constraints among components and their attributes;
- requirement diagrams, which capture general requirements the system specification has to meet;
- behavioral diagrams, which adapt activity diagrams, sequence diagrams, state machine diagrams, and use case diagrams from UML with slight changes to the corresponding activity diagrams in UML.

Roughly speaking, a SysML specification consists of a collection of such diagrams. Neglecting those aspects that refer to relevant information gathered throughout the development process such as comments, viewpoints, rationals underlying design decisions, the underlying conceptual model comprises structural and behavioral concepts. The former ones comprise components and their relationships, which also capture flow information and governing constraints. For the high-level specification of hybrid systems SysML supports the capture of time dependencies by means of continuous, piecewise differentiable functions, though this is handled rather implicitly. The latter ones comprise the state changes resulting from the execution of activities.

As SysML is derived from UML, it inherits its disadvantages [41], mainly the lack of semantics, in particular operational semantics, and the lack of precise indication of the purpose of the various diagrams and how they should be used; SysML provides hardly more than drawing support. Users are left alone with an interpretation and usage of the various diagrams. In particular, with respect to constraints that are decisive for hybrid systems, SysML remains rather vague. A differentiation between synchronous and asynchronous behavior is not available in SysML.

Concerning the supporting tools such as PTC Integrity Modeler [38], Magic Draw [33], Enterprise Architect [24] the situation is even worse. In particular, the latter two shift the emphasis even more towards classical UML modeling, so the applicability for hybrid systems becomes even more obscure. As support for object-oriented programming these tools support a somehow cleansed version of UML, but for hybrid systems engineering they are rather weak.

Concerning formal semantics translations of behavioral diagrams to Abstract State Machines (ASMs) [19] have been developed in [39,44], but these can be merely understood as formalizations of the corresponding UML diagrams, in particular

sequence, activity and statechart diagrams with almost no consideration of the true extensions concerning the integrated continuous parts.

What is required is a more rigorous support for hybrid systems development, which leads directly to methods such as Abstract State Machines, Event-B, TLA $^+$ and others. All these methods have already been used on various occasions for problems associated with hybrid systems. A reasonably good survey of experiences with rigorous methods can be derived from two larger case studies, one addressing the hemodialysis case study [35], which we also address in this article, the other one dealing with the landing gear case study [17], which we also addressed in previous work [21].

Concerning the hemodialysis machine case study Hoang et al. investigate the effectivity of the verification and validation techniques associated with Event-B and the supporting tools based on a formal specification of the safety-critical software in Event-B and an analysis using the RODIN Toolset [28,29]. The HD machine is modeled using iUML-B state machines and class diagrams, and visualized by BMotion Studio, while ProR is used for structuring and tracking requirements. Validation of the model exploits diagrams for modeling sequential properties of the requirements, and ProB-based animation and visualization tools to explore the systems behavior. For the particularly difficult verification of safety properties that involve dynamic behavior co-simulation tools are used to validate against a continuous model of the physical behavior. Arcaini et al. investigate the rigorous development process based on the Abstract State Machine for the hemodialysis machine case study focusing in particular on compliance with standards such as IEC 62304 and general principals of software validation required for safety-critical medical software systems [4,5]. They discuss how to integrate the use of the formal ASM approach emphasizing its refinement principle and model analysis into the current normative for the medical software development. Further contributions were made by Banach [7], Fayolle et al. [25] and Gomes et al. [26]. With the exception of Banach's work, which is based on hybrid Event-B, the major emphasis of all contributions is on the various safety alerts, whereas continuous aspects play a minor role.

Concerning the landing gear case study Su and Abrial propose three Event-B models using different techniques for verification and validation [45]. Another Event-B model by Mammar and Laleau pays particular attention to incremental construction of the model first neglecting time aspects and potential adding these step by step [34]. Ladenberger et al. exploit the B method with particular emphasis on tool-supported visualization [30]. The approach by Arcaini et al. exploits Abstract State Machines emphasizing a refinement-based development, where the correctness of each refinement step is proven by an SMT solver, and code generation in Java [6]. Teodorov et al. address the problem of state space explosion using a context-aware verification approach [47]. All this work merely emphasizes the control and alert aspects in the landing gear case study, whereas the behavior of non-discrete components is not tackled. Though the continuous parts of this case study are not overly complicated—the constraints mainly cover standard relationships between pressure and volume in closed systems such as hydraulic cylinders—Banach makes an attempt to hybrid modeling using hybrid Event-B, though without looking into the resulting verification problems [8].

There are various other case studies on the specification of hybrid systems using formal methods, none of which classifies as complex as the two case studies analysed in more detail. We conclude that while there is a strong need for rigorous support that is not given by methods such as SysML the common rigorous methods still require extensions to become valuable alternatives. This justifies our research in this article as well as in [21], where we provide an enhanced SysML-like approach to structuring and decomposition of hybrid systems preserving the valuable contribution coming from this side, and couple this with a rigorous method, by means of which we enable the accurate specification of asynchronous behavior, the capture of continuous constraints, and the extension of validation and verification approaches including appropriate tool support. Most of this work is still at a rather stage; in particular, hybrid extensions to tools such as RODIN or ASMeta are under development, while the available theory plug-in for real numbers for RODIN [2] is already exploited for dealing with differential equations in hybrid systems [22,23]. We consider it essential that our approach is not bound to a particular method. While we emphasized ASMs in [21], this article is grounded in Event-B. As far as replaceability of one rigorous method by an other one can be carried we like to keep this duality—even extend it to TLA $^+$ and B—and enable choices, unless expressiveness argues in favour of one particular method—for instance, if unbounded parallelism or recursion is to be supported, it is well known that it is only covered by ASMs for the price of complexity.

3. A structural conceptual model for hybrid systems

We will now introduce the concepts for the structural part of our conceptual SE model emphasizing hybrid extensions and dependencies. We will exemplify the model on the hemodialysis machine case study in Section 6.

3.1. Data types

Data types are crucial for the conceptual SE model, as they will be needed for the definition of states. Our model defines complex attributes, which refer to complex data types. For hybrid systems it is decisive that we can handle continuous and differentiable functions on the reals.

Therefore, let b stand for an arbitrary *base type* such as *INT, BOOL, REAL*, etc. We do not impose restrictions on these base types, but we request that *BOOL* and *REAL* are always in the given collection of base types. Then $\hat{b}$ denotes an extension of the base type b by the value *undef*, which is necessary for partial functions. For the base types we assume a predefined set

of operators such as the Boolean operators $\neg$, $\wedge$ and $\vee$ (and any shortcut), addition + , multiplication $\cdot$, etc. for integers and reals, etc.

Definition 3.1. *Data types* t are defined by the following abstract syntax:

$$t = b \mid (a_1 : t_1, \ldots, a_n : t_n) \mid \{t\} \mid \langle t \rangle \mid [t] \mid (a_1 : t_1) \uplus \cdots \uplus (a_n : t_n) \mid t \to t'$$

As usual $(a_1 : t_1, \ldots, a_n : t_n)$ denotes a record type constructor with component types $t_1, \ldots, t_n$ with labels $a_1, \ldots, a_n$, $\{t\}$ denotes a constructor for finite sets with elements of type t, $\langle t \rangle$ denotes a constructor for finite multisets with elements of type t, $[t]$ denotes a constructor for finite lists with elements of type t, $(a_1 : t_1) \uplus \cdots \uplus (a_n : t_n)$ denotes a constructor for disjoint unions of component types $t_1, \ldots, t_n$ with labels $a_1, \ldots, a_n$, and $t \to t'$ denotes a constructor for the type of continuous functions defined on values of type t and resulting in values of type t'. Here we use the usual topology for the reals and product topologies, whenever this is applicable, as e.g. for the case of record types. In all other cases the topology is the discrete topology.

Example 3.1. In Section 6 we define several data types for the hemodialysis machine case study. In particular, following (6.2) we define the type $Pressure = REAL \to REAL$ capturing pressure values that depend on time. Similarly we define $Volume = REAL \to REAL$ capturing volume that can be manipulated over time. We further need a type $YesNo = (\text{yes} : \mathbb{1}) \uplus (\text{no} : \mathbb{1})$ in all three cases, which actually takes exactly two values. These types are also used after (6.3).

For all constructed types we use the usual operators such as projections for records, structural recursion for sets, multisets and lists, selection for union types, and concatenation, restriction, etc. for function types. Such operators are defined in detail in [40]. In addition, we require a derivation operator $\mathcal{D}_t = \dfrac{\partial}{\partial t}$ that is defined for (partial) functions with domain *REAL*. Naturally, $\mathcal{D}_t(f)$ is also a partial function with domain *REAL*, and $\mathcal{D}_t(f)(x)$ is the derivative of f at the point x (sometimes also denoted as $f'(x)$), provided this exists. Thus, $\mathcal{D}_t$ is actually a functional.

Each data type t defines a set of values, which we denote as $dom(t)$. This can be exploited to represent defined data types in an Event-B context, while theories comprising types, operators and axioms are used to define base types such as *Bool* or *Real*. The report [22] contains very detailed definitions of such theories, as they are used in [23] and supported by the RODIN theory plug-in [2]. The work in [14] further shows how to realize exact real arithmetic.

For later use we also permit *subtyping*.

Definition 3.2. *Subtypes* are defined as follows:

- each type is a subtype of itself and of the base type $\mathbb{1}$ with $dom(\mathbb{1}) = \{\text{undef}\}$;
- each type $(a_{i_1} : t'_{i_1}, \ldots, a_{i_m} : t'_{i_m})$, where $\{i_1, \ldots, i_m\} \subseteq \{1, \ldots, n\}$ and t'_i is a subtype of t_i ($i \in \{i_1, \ldots, i_m\}$), is a subtype of $(a_1 : t_1, \ldots, a_n : t_n)$;
- each type $\{t'\}$, where t' is a subtype of t, is a subtype of $\{t\}$;
- each type $\langle t' \rangle$, where t' is a subtype of t, is a subtype of $\langle t \rangle$;
- each type $[t']$, where t' is a subtype of t, is a subtype of $[t]$;
- each type $(a_1 : t'_1) \uplus \cdots \uplus (a_n : t'_n)$, where t'_i is a subtype of t_i ($i \in \{1, \ldots, n\}$), is a subtype of $(a_1 : t_1) \uplus \cdots \uplus (a_n : t_n)$;
- each type $t_1 \to t'_1$, where t is a subtype of t_1 and t'_1 is a subtype of t', is a subtype of $t \to t'$.

Whenever t' is a subtype of t we obtain a canonical *subtype function* $\pi : dom(t) \to dom(t')$. Such subtype functions can be defined in Event-B contexts.

3.2. Blocks and structures

The structural building blocks of our conceptual model are block types and clusters of different order $n \geq 0$, which we derive from HERM. It can well be argued (see e.g. [27] or [19]) that any mathematical theory is grounded in a first-order Tarski structure (aka universal algebra), i.e. a set of functions, and functions can be represented by relations and vice versa.

Definition 3.3. A *block type* B of order $n \geq 0$ consists of a set $\{r_1 : B_1, \ldots, r_k : B_k\}$ of pairwise different (labelled) *components* and a set $\{A_1, \ldots, A_\ell\}$ of *attributes*. The labels $r_i \in \mathcal{L}$ are taken from a set $\mathcal{L}$ of labels, also called *roles*. Each component B_i is a block type or a cluster of order $n_i < n$, and for at least one component the order must be exactly $n - 1$. Each attribute A_i is associated with a *data type* $\text{type}(A_i)$.

A *cluster* C of order $n \geq 0$ consists of a labelled set $\{c_1 : B_1, \ldots, c_m : B_m\}$ with pairwise different labels $c_i \in \mathcal{L}$, where each component B_i is a block type of order $n_i \leq n$, and for at least one component the order must be exactly n.

Each block type defines its parts and describing properties. As block types of order 0 do not have components, we call them *elementary*.

The intended semantics is easily explained. A block type gives rise to a relation, and each cluster to a disjoint (labelled) union of relations. More precisely, if B is a block type as in Definition 3.3, then a B-tuple is a record $(id : i, r_1 : i_1, \ldots, r_k : i_k, A_1 : v_1, \ldots, A_\ell : v_\ell)$ of arity $k + \ell + 1$ (k is the number of components, ℓ the number of attributes), where $i, i_1, \ldots, i_k$ are identifiers taken from some not further specified set *ID* of identifiers, and each v_i is a value of type $\text{type}(A_i)$.

Example 3.2. In Section 6 we define a schema comprising several block types. For instance, the definition of a block type HD_Machine in (6.1) comprises four components and no attributes. The definition of the elementary block type Control_Subsystem in (6.2) comprises nine attributes. The definition of a block type EBC in (6.3) comprises six components and again no attributes. Further block types are defined in 6.4 and 6.5 and the text following these definitions.

According to these definitions block type HD_Machine has order 2, block types EBC, Dialyser and DF_Subsystem have order 1 and all other block types have order 0, i.e. they are elementary.

Definition 3.4. A *schema* is a set $\{B_1, \ldots, B_n\}$ of block types and clusters that is closed under components. A *structure* $\mathcal{S}$ associates a set $\mathcal{B}$ of B-tuples with each block type B of the schema, and the labelled disjoint union $\mathcal{C} = \bigcup_{i=1}^{m} \{(c_i, t_i) \mid t_i \in \mathcal{B}_i\}$ with each cluster B. The structure $\mathcal{S}t$ is *well-defined* iff all identifiers associated to tuples are pairwise different, and whenever r_j: i_j appears in a B-tuple in $\mathcal{B}$, then there exists an B_j-tuple in $\mathcal{B}_j$ with identifier i_j.

We only consider well-defined schemata and structures in this article. According to Definition 3.4 each block type B gives rise to set values $\mathcal{S}(B)$. Therefore, in Event-B we can employ variables B and invariants capturing the type of $\mathcal{S}(B)$. Furthermore, the well-definedness conditions give rise to further invariant clauses.

Example 3.3. All definitions of block types in (6.1)–(6.5) and the accompanying text together define a well-defined schema.

If $\mathcal{S}$ is a well-defined structure, B is a block type and $i \in ID$ is an identifier, then $B(i)$ denotes the B-tuple in the structure with identifier i. Then $B(i).r_j$ (for $j = 1, \ldots, k$) and $B(i).A_j$ (for $j = 1, \ldots, \ell$), are used to denote the value of the corresponding component, i.e. $B(i).r_j \in ID$, or attribute, i.e. $B(i).A_j \in \text{type}(A_i)$, respectively. As identifiers are of minor importance (except for referencing and unique identification), we also use the notation $B(i)!r_j$ (for $j = 1, \ldots, k$) for the B_j-tuple with identifier $B(i).r_j$. In doing so, we can further use paths $\wp = B_0, \ldots, B_n$ with $B_0 = B$ and let $B(i)\wp$ denote the B_n-tuple that is reached from $B(i)$ through this path.

Pragmatically, in systems engineering we deal with devices that are structured into components, so there is usually a *root* block type B_r, from which all attributes of all components can be reached by some path. If we fix an identifier $i \in ID$, then any attribute in any component can be denoted by some expression $B_r(i)\wp A_j$. Instead of this lengthy notation we permit shortcuts: if the attribute name A_j is unique and there is only one path $\wp$ from B_r to it, then we simply use A_j, otherwise use a shortcut name $\wp'$ for $B_r(i)\wp$ and write $\wp'\_A_j$.

A schema can be easily represented by a diagram, i.e. a labelled graph. If $\mathcal{S}$ is a schema, then we define a directed graph (V, E), the *diagram* representing $\mathcal{S}$, as follows:

- The set V of vertices comprises four types:
  - each block type B of order 0 defines a vertex labelled by B (usually represented by a rectangle);
  - each block type B of order $n > 0$ defines a vertex labelled by B (usually represented by a diamond);
  - each cluster C defines a vertex labelled by C (usually represented by a circle with $\oplus$ symbol);
  - each attribute A of a block type B defines a vertex labelled by A and optionally also by the associated type type(A).
- The set E of directed edges comprises three types:
  - role edges from a block type B to each of its components r: B' labelled by the role $r \in \mathcal{L}$;
  - component edges from a cluster C to each of its components c: B labelled by $c \in \mathcal{L}$;
  - attribute edges from a block type B to each of its attributes A.

Example 3.4. The schema of the hemodialysis machine case study is illustrated by a diagram that is given by the combination of the subdiagrams in Figs. 6.2–6.5.

For later use the nested character of complex-valued attributes by means of the data types permit the definition of subattributes. If A is an attribute, then any subtype t' of $t = \text{type}(A)$ defines a *subattribute* $A' = \pi(A)$ with $\text{type}(A') = t'$, where π is the canonical subtype function $dom(t) \to dom(t')$.

For the sake of keeping diagrams reasonably small we permit using *subdiagrams*. For these omit components or attributes from types in a schema $\mathcal{S}$ or even omit types completely, as long as the well-definedness is still guaranteed. The diagram of the resulting subschema constitutes a subdiagram. However, even if all components of a block type B were removed in such a subdiagram construction, the order of B will still be defined by the complete schema. We say that a set of subdiagrams *covers* $\mathcal{S}$ iff each of the vertices and edges of the diagram of $\mathcal{S}$ appears in at least one subdiagram.

Example 3.5. Each of the Figs. 6.2–6.5 illustrates a subdiagram of the diagram for the hemodialysis machine case study.

4. Constraints

HERM provides a large variety of possible constraints that can be used to restrict the admitted structures. As our model is based on HERM, we inherit this richness to express constraints, which we call structural dependencies. We will further extend them by flow dependencies and parametric constraints that are of particular importance for the hybrid extensions. For the semantics of our conceptual model we have to guarantee that all structures are valid. For examples of the constraints we refer to Section 6.

Definition 4.1. A structure is called *valid* with respect to a set Σ of constraints iff it satisfies all constraints in Σ.

4.1. Structural dependencies

Important classes of static constraints are key and functional dependencies, inclusion and exclusion dependencies, cardinality constraints and path constraints.

Definition 4.2. A *key* for a block type $B = (\{r_1 : B_1, \ldots, r_k : B_k\}, \{A_1, \ldots, A_\ell\})$ is defined by $K = (\{r_{i_1} : B_{i_1}, \ldots, r_{i_m} : B_{i_p}\}, \{A'_{j_1}, \ldots, A'_{j_q}\})$ comprising a subset of the set of components and a set of subattributes A'_{j_i} of A_{j_i}.

As K gives rise to a canonical projection π_K of B-tuples to K-tuples exploiting the canonical subtype functions for subattributes, a structure $\mathcal{S}$ *satisfies the key dependency* defined by K iff $\pi_K(v_1) = \pi_K(v_2) \Rightarrow v_1 = v_2$ holds for all B-tuples $v_1, v_2 \in \mathcal{S}(B)$.

Definition 4.3. If X and Y are built in the same way as K for a block type B, then the expression $X \to Y$ defines a *functional dependency*.

A structure $\mathcal{S}$ *satisfies a functional dependency* $X \to Y$ iff $\pi_X(v_1) = \pi_X(v_2) \Rightarrow \pi_Y(v_1) = \pi_Y(v_2)$ holds for all B-tuples $v_1, v_2 \in \mathcal{S}(B)$.

Definition 4.4. If X_1 and X_2 are built in the same way as K for block types B_1 and B_2, respectively, and both π_{X_i} map B_i-tuples to tuples of the same type, then the expression $B_1[X_1] \subseteq B_2[X_2]$ defines an *inclusion dependency*. Analogously, an expression $B_1[X_1] \| B_2[X_2]$ defines an *exclusion dependency*.

The *validity of an inclusion dependency* $B_1[X_1] \subseteq B_2[X_2]$ in a structure $\mathcal{S}$ requires that for each B_1-tuple $v_1 \in \mathcal{S}(B_1)$ there exists a B_2-tuple $v_2 \in \mathcal{S}(B_2)$ with $\pi_{X_1}(v_1) = \pi_{X_2}(v_2)$. The *validity of an exclusion dependency* $B_1[X_1] \| B_2[X_2]$ in a structure $\mathcal{S}$ requires that for all B_i-tuples $v_i \in \mathcal{S}(B_i)$ we have $\pi_{X_1}(v_1) \neq \pi_{X_2}(v_2)$.

Definition 4.5. For a block type B with components $r_{i_1} : B_{i_1}, \ldots, r_{i_k} : B_{i_k}$ and a subset $I \subseteq \mathbb{N}$ the expression $\mathrm{card}(B, (r_{i_1} : B_{i_1}, \ldots, r_{i_k} : B_{i_k})) = I$ defines a *participation cardinality constraint*.

Analogously, for $r_{j_1} : B_{j_1}, \ldots, r_{j_\ell} : B_{j_\ell}$ and $I \subseteq \mathbb{N}$ the expression $\mathrm{look}((r_{i_1} : B_{i_1}, \ldots, r_{i_k} : B_{i_k}), (r_{j_1} : B_{j_1}, \ldots, r_{j_\ell} : B_{j_\ell})) = I$ defines a (k, ℓ) *look-up cardinality constraint*.

In both cases I is most commonly an interval $[m, M]$. A structure $\mathcal{S}$ *satisfies a participation cardinality constraint* $\mathrm{card}(B, (r_{i_1} : B_{i_1}, \ldots, r_{i_k} : B_{i_k})) = I$ iff for all B_i-tuples $v_i \in \mathcal{S}(B_i)$ $(i \in \{i_1, \ldots, i_k\})$ with identifiers id_i the set of B-tuples $\{v \in \mathcal{S}(B) \mid \pi_{\{r_{i_1}:B_{i_1},\ldots,r_{i_k}:B_{i_k}\}}(v) = (id_{i_1}, \ldots, id_{i_k})\}$ has a cardinality in I.

Analogously, a structure $\mathcal{S}$ *satisfies a look-up cardinality constraint* $\mathrm{look}((r_{i_1} : B_{i_1}, \ldots, r_{i_k} : B_{i_k}), (r_{j_1} : B_{j_1}, \ldots, r_{j_\ell} : B_{j_\ell})) = I$ iff for for all B_i-tuples $v_i \in \mathcal{S}(B_i)$ $(i \in \{i_1, \ldots, i_k\})$ with identifiers id_i the set $\{\pi_{\{r_{j_1}:B_{j_1},\ldots,r_{j_\ell}:B_{j_\ell}\}}(v) \mid v \in \mathcal{S}(B) \wedge \pi_{\{r_{i_1}:B_{i_1},\ldots,r_{i_k}:B_{i_k}\}}(v) = (id_{i_1}, \ldots, id_{i_k})\}$ has a cardinality in I.

Though in Section 6 we do not go into details of "classical" dependencies as defined above, we mention participation cardinality constraints following block definitions, e.g. in (6.1), which arise from the fact that the components are in fact parts.

All kinds of dependencies defined above can be generalized to *path dependencies*, if the block types used in their definitions are expanded types $B_{\wp}$ according to paths $\wp$

Definition 4.6. A *path* is a sequence $\wp = B_1, \ldots, B_k$ of block types or clusters such that B_{i+1} appears as a component of B_i for all $i = 0, \ldots, k-1$.

If we replace a component $r_{i+1} : B_{i+1}$ in B_i by $r_{i+1,j} : B'_j$ for all components $r'_j : B'_j$ of B_{i+1} and add all attributes of B_{i+1} to those of B_i, we obtain an expanded block type $B_{i,i+1}$. Doing this for all components in a path defines an expanded block type $B_{\wp}$.

With the definition of semantics for the structural dependencies in Definitions 4.2–4.5 and the extended path constraints it becomes obvious that all these constraints give rise to further invariant clauses in representing Event-B machines.

4.2. Flow dependencies and parametric constraints

With respect to attributes that are bound to types of continuous functions we introduce *flow dependencies* and *parametric constraints*. The former ones indicate that a value from one component may flow into another component. In order to capture such dependencies in our conceptual model we permit some of the attributes of a block type to be declared as *ports*. These can be *in*-ports, *out*-ports and *in-out*-ports.

Definition 4.7. A *flow dependency* links an out-port of one block type with an in-port of another one, or two in-out-ports.

By default, if nothing else is specified, a flow dependency indicates that at any time the value associated with the connected attributes are equal. That is, flow dependencies are technically just specific path dependencies. They are particularly relevant, if the associated data type, which must be the same for both ports, is $\mathbb{R} \to t$. By means of a flow dependency a synchronization of the values in a structure will be enforced. This will become relevant in the behavioral part of our model.

Example 4.1. In (6.6)–(6.9) we give several examples of flow dependencies for the hemodialysis machine case study. Some of these are linked to real flows of liquids, others to connections from sensors to the control component. It will be a matter of refinement to highlight the type of flow interaction. Note that a flow dependency always implies inclusion dependencies.

The second class of dependencies that are not inherited from HERM are *parametric constraints* that are used to express inherent constraints in the physical domain such as dependencies between pressure and volume or requirements concerning bounds of a continuous functions, etc. Parametric constraints are usually associated with continuous functions, though our definition permits a more general usage.

Definition 4.8. A *parametric constraint* is given by an equation or inequality with n variables $x_1, \ldots, x_n$, each of which is linked to an attribute in some block type.

A parametric constraint may involve the derivation functionals $\mathcal{D}_t$, by means of which differential equations, in particular solutions to initial value problems, can be requested as constraints. Same as the flow dependencies a synchronization of the values in a structure will be enforced.

Example 4.2. The hemodialysis machine case study in Section 6 contains many parametric constraints, in particular those defined in (6.10)–(6.14). Some of these formalize common physical laws, e.g. (6.12) formalizes es the relationship between pressure and volume in a closed system. Others define conditions the system must satisfy; otherwise an alert will be produced.

Parametric constraints can be combined with flow dependencies, which defines parametric flow dependencies.

Definition 4.9. A *parametric flow dependency* is a flow dependency together with a parametric constraint on two variables x_{in}, x_{out} (or x_1, x_2, respectively) representing the in- and out-port of the dependency (or the two in-out-ports, respectively).

Technically, a parametric flow dependency is again a path dependency, which enforces that whenever x_{in} takes a particular value, that x_{out} takes a value determined by the parametric constraint. The flow is always from the in-port to the out-port. It is bi-directional only in case of in-out-ports. Parametric flow dependencies may e.g. be used to specify time delays.

Same as for the structural dependencies all flow dependencies and parametric constraints can be captured by invariants in Event-B.

5. A behavioral conceptual model for hybrid systems

For the semantics of the structural part we exploit Event-B, more precisely multiple Event-B machines with a single context. The context captures the data type definitions from the structural model as explained in Section 3. As each data type defines a set of values, the context specifies these sets explicitly.

5.1. Multiple concurrent Event-B machines

Event-B machines define variables, invariants and events [1]. In a nutshell an Event-B machine comprises a finite set $\mathcal{V}$ of *state variables*, an *invariant* $\mathcal{I}$ and a finite set $\mathcal{E}$ of *events*. One of the events is an *initialization* event *init*. We may assume a *universe* $\mathcal{U}$ of values and several pre-defined domains $\mathcal{D}_i \subseteq \mathcal{U}$ such as domains of integers, Booleans, real numbers, character strings, etc. For all these domains we may further assume pre-defined operations such as addition, multiplication, concatenation, etc.

An Event-B machine is bound to a *context*, in which further sets and further operations on them can be defined. For these the usual constructors for sets such as comprehension, products, unions, etc. as well as λ-abstraction to define functions can be exploited. Using constants, operations and predicates provided by the context, the state variables (treated as constants) plus other variables we may define *terms* and first-order *formulae* in the usual way. The invariant $\mathcal{I}$ must be a closed formula defined in this way.

A *state* is defined by assigning a value in $\mathcal{U}$ to each state variable $x \in \mathcal{V}$. We write $\mathrm{val}_S(x)$ for the value assigned to x in state S. We may interpret terms and formulae in a state S and extend the evaluation function val in this way. If t is a term, say $t = f(t_1, \ldots, t_n)$, then $\mathrm{val}_S(f(t_1, \ldots, t_n)) = f(\mathrm{val}_S(t_1), \ldots, \mathrm{val}_S(t_n))$. Note that f on the write-hand-side of this equation denotes the pre-defined operation f. Clearly, the evaluation of a term that is a state variable x is given by its value in the state. If x is an arbitrary variable, not a state variable, then its interpretation requires a *variable assignment*, i.e. a function σ from the set of such variables to $\mathcal{U}$, and we have $\mathrm{val}_S(x) = \sigma(x)$.

Atomic formulae are interpreted in the same way resulting in a truth value **true** or **false**. This is then extended in the usual way for negation, conjunction, disjunction, implication and quantified formulae. In particular, as $\mathcal{I}$ is a closed formulae, $\mathrm{val}_S(\mathcal{I})$ denotes a truth value. As usual, we write $S \models \mathcal{I}$ iff $\mathrm{val}_S(\mathcal{I}) = \mathbf{true}$ holds, in which case we say that $\mathcal{I}$ *is satisfied* in state S. A state S is a *valid state* iff $\mathcal{I}$ is satisfied in S.

Each event $e \in \mathcal{E}$ takes the form

$$\mathbf{ANY}\ y_1, \ldots, y_n\ \mathbf{WHEN}\ \varphi\ \mathbf{THEN}\ x_1 := t_1 \| \ldots \| x_m := t_m,$$

i.e. it is an unbounded choice with a parallel assignment. Here $y_1, \ldots, y_n$ are arbitrary variables, φ is a formula that contains at most $y_1, \ldots, y_n$ as free variables, $x_1, \ldots, x_m$ are the state variables, and $t_1, \ldots, t_m$ are terms that may use the variables $y_1, \ldots, y_n$.

An event e is *enabled* in state S iff for some assignment of values to the variables $y_1, \ldots, y_n$ the guard φ holds[3]. An enabled event may fire[4], which results in a *successor state* S' of S. Informally, choose a variable assignment σ such that φ is satisfied in S using this variable assignment, then determine $v_i = \mathrm{val}_S(t_i)$ using σ for the interpretation of the free variables $y_1, \ldots, y_n$ and let $\mathrm{val}_{S'}(x_i) = v_i$.

Then a *run* of the machine[5] is a sequence $S_0, S_1, S_2, \ldots$ of valid states such that S_0 is an initial state and for each i the state S_{i+1} is a successor state of S_i resulting from firing events that is enabled in S_i. Note that this definition of run is highly non-deterministic, as the unbounded choice allows us to select values for the variables $y_1, \ldots, y_n$, which determine the new values that are assigned to the state variables.

In Sections 3 and 4 we explained how we obtain variables and invariants from the structural conceptual model. However, we cannot simply take all these variables and invariants into a single Event-B machine and complement them by the definition of events:

- For the definition of the semantics of Event-B it is assumed that when conditions of several events are simultaneously satisfied, the events are executed in parallel in a synchronous way. As already explained in the introduction, the proof of Gurevich's behavioral theory of sequential algorithms [27] implies that any sequential algorithm can be captured step-by-step by an equivalent Event-B machine. That is, while bounded synchronous parallelism is well captured, hybrid systems with multiple components require several events to be fired concurrently in an asynchronous way. For this we have to exploit multiple Event-B machines, for which an appropriate semantics has to be defined[6].
- When multiple machines are introduced adopting the semantics of concurrency as developed in the context of ASMs [18], the interaction of the various machines will require shared memory locations, which are also uncommon for Event-B.

For the extension of Event-B to a concurrent multiple machine model we exploit that if $S_0, S_1, S_2, \ldots$ is a run of an Event-B machine with state variables $\mathcal{V} = \{x_1, \ldots, x_m\}$, then we obtain *differences* between a state S_i and its successor S_{i+1}. For this let $\mathrm{Diff}_i = \{x \in \mathcal{V} \mid \mathrm{val}_{S_i}(x) \neq \mathrm{val}_{S_{i+1}}(x)\}$ be the set of state variables, on which the state S_i and its successor S_{i+1} differ. Then define the *difference set* (or *update set* $\Delta_i = \{(x, \mathrm{val}_{S_{i+1}}(x)) \mid x \in \mathrm{Diff}_i\}$. This is called update set, because the change from S_i to S_{i+1} updates exactly the state variables in Diff_i to their new value given in Δ_i. Correspondingly, each element of Δ_i, i.e. a pair comprising a state variable and a value, is called an *update*.

If Δ is an arbitrary set of updates on a state S, then $S + \Delta$ denotes the state resulting from S by applying the update set to it. For each state variable x we have

$$\mathrm{val}_{S+\Delta}(x) = \begin{cases} v & \text{if } (x, v) \in \Delta \text{ and } \Delta \text{ is consistent} \\ \mathrm{val}_S(x) & \text{otherwise} \end{cases}.$$

An update set is called *consistent* iff for all $x \in \mathcal{V}$ and all $v_1, v_2 \in \mathcal{U}$ with $(x, v_1) \in \Delta$ and $(x, v_2) \in \Delta$ we have $v_1 = v_2$. Note that the update set defined by the difference of two states in a run is always consistent, and we have $S_{i+1} = S_i + \Delta_i$.

Phrased differently, each run of an Event-B machine results from building update sets and applying them. For a rule of an event e the possible update sets in state S take the form $\{(x_i, \mathrm{val}_S(t_i)\}$. As the values resulting from the interpretation of the terms t_i depend on the variable assignment for the variable y_i and further the event be must be enabled, we define

$$\mathbf{\Delta}_i = \{\{(x_j, \mathrm{val}_{S_i}(t_j)\} \mid S_i \models \varphi(y_1, \ldots, y_n) \wedge \mathrm{grd}(e)\}.$$

$\mathbf{\Delta}_i$ is the set of possible update sets of the Event-B machine in state S_i. Thus in a run $S_0, S_1, S_2, \ldots$ we always have $S_{i+1} = S_i + \Delta$ for some $\Delta \in \mathbf{\Delta}_i$.

This allows us to proceed from single Event-B machines to multiple machines. For this take a finite family $\{\mathcal{M}_i \mid i \in I\}$ of Event-B machines $\mathcal{M}_i$ (for convenience we use some finite index set I here). For the semantics the key idea is to separate the building of update sets from their application, by means of which we take care of the different pace, with which the autonomous machines $\mathcal{M}_i$ operate. This reflects the essential property of asynchronously collaborating machines of having each its own clock according to which they make steps.

Let $\mathcal{V} = \bigcup_{i \in I} \mathcal{V}_i$ be the union of the sets of state variables of the machines $\mathcal{M}_i$. A state S defined over $\mathcal{V}$ is called *global*, whereas the restricted states $\mathrm{res}_i(S)$ built over $\mathcal{V}_i$ are called *local* for $\mathcal{M}_i$. While $\mathrm{res}_i(S)$ is valid iff it satisfies the local invariant $\mathcal{I}_i$ associated with the machine $\mathcal{M}_i$, a global state S is *valid* iff $S \models \bigwedge_{i \in I} \mathcal{I}_i$ holds.

Then, in analogy to the definition in [18] we define a concurrent system of Event-B machines as an I-indexed family $\mathcal{M} = \{\mathcal{M}_i\}_{i \in I}$ of Event-B machines $\mathcal{M}_i$ such that some variables are shared between several machines. The semantics is defined by concurrent runs.

---

[3] Note that this reflects the reactive behavior of events in Event-B. The guard is observed, and only when it becomes satisfied, the parallel assignment come into action.

[4] Note that in Event-B only one event that is enabled in a state is selected for firing.

[5] Note that runs are an immediate consequence of the state transitions defined by the events. There are usually not emphasized in the Event-B literature, as invariance and refinement proofs only require single events to be considered. However, for the concurrent extension below runs will become more important.

[6] Let us emphasize here again that this is indeed an extension to Event-B.

Definition 5.1. Let $\mathcal{M} = \{\mathcal{M}_i\}_{i\in I}$ be a concurrent system of Event-B machines. A *concurrent run* of $\{\mathcal{M}_i \mid i \in I\}$ is a sequence $S_0, S_1, S_2, \ldots$ of valid global states, in which S_{i+1} is a successor state of S_i.

For this definition it remains to clarify the notion of successor state of a global state. For this consider first finite subsets $I_i \subseteq I$ of indices of those machines that initiate a step in S_i. For each $j \in I_i$ we obtain a set $\mathbf{\Delta}_{i,j}$ of possible update sets of $\mathcal{M}_i$ in the local state $\mathrm{res}_j(S_i)$ arising by restriction from the global state S_i.

For the transition from the global state S_i to a successor S_{i+1} we take another finite subset $\hat{I}_i \subseteq I$ of indices of those machines that finish their step in S_{i+1}. Then for each $j \in \hat{I}_i$ there exists an index $k(j) \leq i$ such that the step of $\mathcal{M}_j$ was initiated in state $S_{k(j)}$. So we obtain an update set $\Delta_i = \bigcup_{j\in\hat{I}_i} \Delta_{k(j),j}$ selecting update sets $\Delta_{k(j),j} \in \mathbf{\Delta}_{k(j),j}$, with which we can define the successor state $S_{i+1} = S_i + \Delta_i$.

Informally phrased, in a concurrent run the sequence of global states results from simultaneously applying update sets of several individual machines that have been built on previous (not necessarily the last nor the same) states. Note that the definition leaves it completely open, how big the difference between i and $k(j)$ is, which reflects that a machine may operate slowly or fast, but is completely oblivious to the activities of the other machines in the concurrent family. This adds another source of non-determinism; even if the individual machines operate deterministically, the multi-machine family will not.

Further note that the asynchronous parallel behavior of the machines $\mathcal{M}_i$ in the family as expressed by the definition of concurrent runs does not rely on interleaving, but permits simultaneous updates by several machines. It is not a weakness but a strength of state-based methods that any mimicking of collaboration in parallel[7] by means of interleaving can be dispensed with.

Thus, we will capture the behavior of a hybrid by a concurrent system of Event-B machines subject to this definition of concurrent runs. Each of the machines involved in this collection will comprise some of the variables determined by the structural conceptual model and the invariants associated with these variables as determined by the constraints in the structural conceptual model. A variable is pliant, if its type involves the continuous function type constructor; otherwise it is mode. All variables and constraints as derived in the previous sections will have to be captured in at least one machine $\mathcal{M}_i$; variables that appear in more than one machine are shared.

While the variables of the machines are linked to the attributes of the components in the structural model, these two notions have to separated carefully. As machines interact asynchronously, whereas flow dependencies enforce synchronous behavior, the variables associated with each Event-B machine $\mathcal{M}_i$ in a concurrent system correspond to a view.

Definition 5.2. A *view schema* $\mathcal{S}_v$ is another schema that results from $\mathcal{S}$ by omitting block types or clusters B_i, omitting components or attributes of a block type B_i, and replacing any attribute A_j of a block type B_i by a subattribute.

5.2. Separation of concerns

The structural conceptual model that we developed in Section 3 emphasizes the decomposition of a hybrid system into components, for which we introduced block types. These block types are associated each with an order $n \geq 0$. The behavior of elementary block types of order 0 can only be either continuous or discrete, but not hybrid. Otherwise further structuring into components should be possible. This reflects the fact that in reality continuous behavior is associated with a piece of hardware in a general sense subsuming electrical, mechanical and hydraulic parts, whereas discrete behavior is associated with a piece of software. We therefore claim that in a sufficiently refined model continuous and discrete aspects must be separated on the level of elementary block types.

Consequently, there is a need for Event-B machines that capture the behavior of continuous components. Flow dependencies (or parametric flow dependencies) guarantee that the manipulation of control variables can be done by discrete Event-B machines, while the (parametric) flow dependency itself is assured by an assertion.

Definition 5.3. An *assertion* is a statement that a flow dependency or parametric flow dependency is always satisfied.

Such an assertion can be grounded in a general law of physics or it can be confirmed by experiments with the continuous component associated with the flow. Alternatively, it is possible to define a distinguished Event-B machine that maintains the flow dependency. In this case, whenever the in-port (or one of the in-out-ports) of the flow dependency is updated, an update to the out-port (or the other in-out-port) is triggered enforcing the flow dependency to be satisfied.

6. The hemodialysis machine case study

In this section we apply our conceptual model to the hemodialysis machine case study described in [35]. This will illustrate the key modeling ideas, though the model itself will necessarily remain incomplete.

A hemodialysis machine (as illustrated schematically in Fig. 6.1) is used as an artificial filtration system that extracorporally purifies the blood by removing impurities and unwanted fluids. A typical hemodialysis procedure is performed in the following fashion:

[7] In fact, interleaving expresses parallelism by sequentialization, which is not exactly what happens in reality. For systems with a sequential implementation—this includes all those built at the time the notion of interleaving was invented—this may be acceptable, for truly asynchronous systems—this includes all distributed systems with multiple processors spread over a network—this workaround is not needed, but in contrast counter-productive.

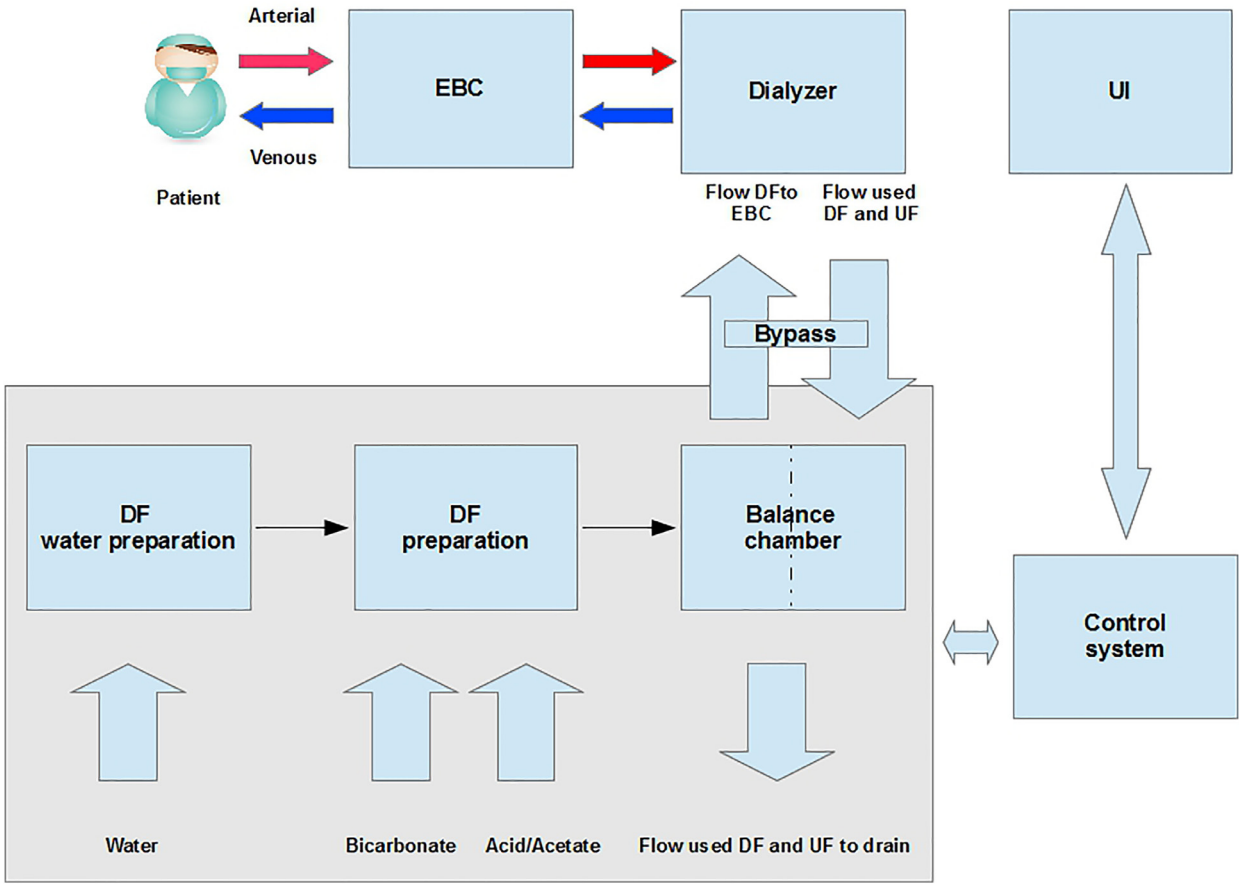

Fig. 6.1. Schematic view of a hemodialysis machine [35].

1. Two needles are inserted into a patient's arm, one to take out the blood and the other to put the blood back. A specific amount of blood (approximately five percent of the patient's blood) is taken outside the body at a given time.
2. The blood is taken through the arterial access of the patient's arm. The blood then travels through a thin tube that takes the blood to the dialyzer. A dialyzer is comprised of many fine hollow fibers which are made of a semi-permeable membrane.
3. As the blood flows through these fibers, the dialysate (a chemical substance) flows around them, removing impurities and excess water and adjusting the chemical balance of the blood.
4. After being cleansed and treated, the blood is returned to the patient's arm through the venous access.

6.1. Component model of a hemodialysis machine

We first describe a schema comprising the components of a hemodialysis machine. We have to omit some details to keep the illustration of the conceptual model reasonably compact. Furthermore, the model will already be rather refined, as we do not emphasize the development process. The schema is depicted by the combination of the four subdiagrams in Figs. 6.2–6.5.

As shown in Fig. 6.2 a hemodialysis machine comprises four components: the extracorporal blood circuit (EBC), the dialyser component (DIALYSER), the dialyser fluid subsystem (DF_SUBSYSTEM), and the control unit (CONTROL_SUBSYSTEM). So the definition of the block type HD_MACHINE (following Definition 3.3) takes the form

$$(\{\text{control} : \textsc{Control\_Subsystem}, \text{dialyser} : \textsc{Dialyser}, \\ \text{ebc} : \text{EBC}, \text{df} : \textsc{DF\_Subsystem}\}, \emptyset). \tag{6.1}$$

Note that for all four components we have participation cardinality constraints with value 1, i.e. each dialyser or control component belongs to exactly one hemodialysis machine.

The component CONTROL_SUBSYSTEM is defined by an elementary block type (see Definition 3.3) of order 0 with attributes shown in Fig. 6.2. That is we have

$$(\emptyset, \{\text{arterial\_pressure}, \text{venous\_pressure}, \text{arterial\_entry\_pressure},$$

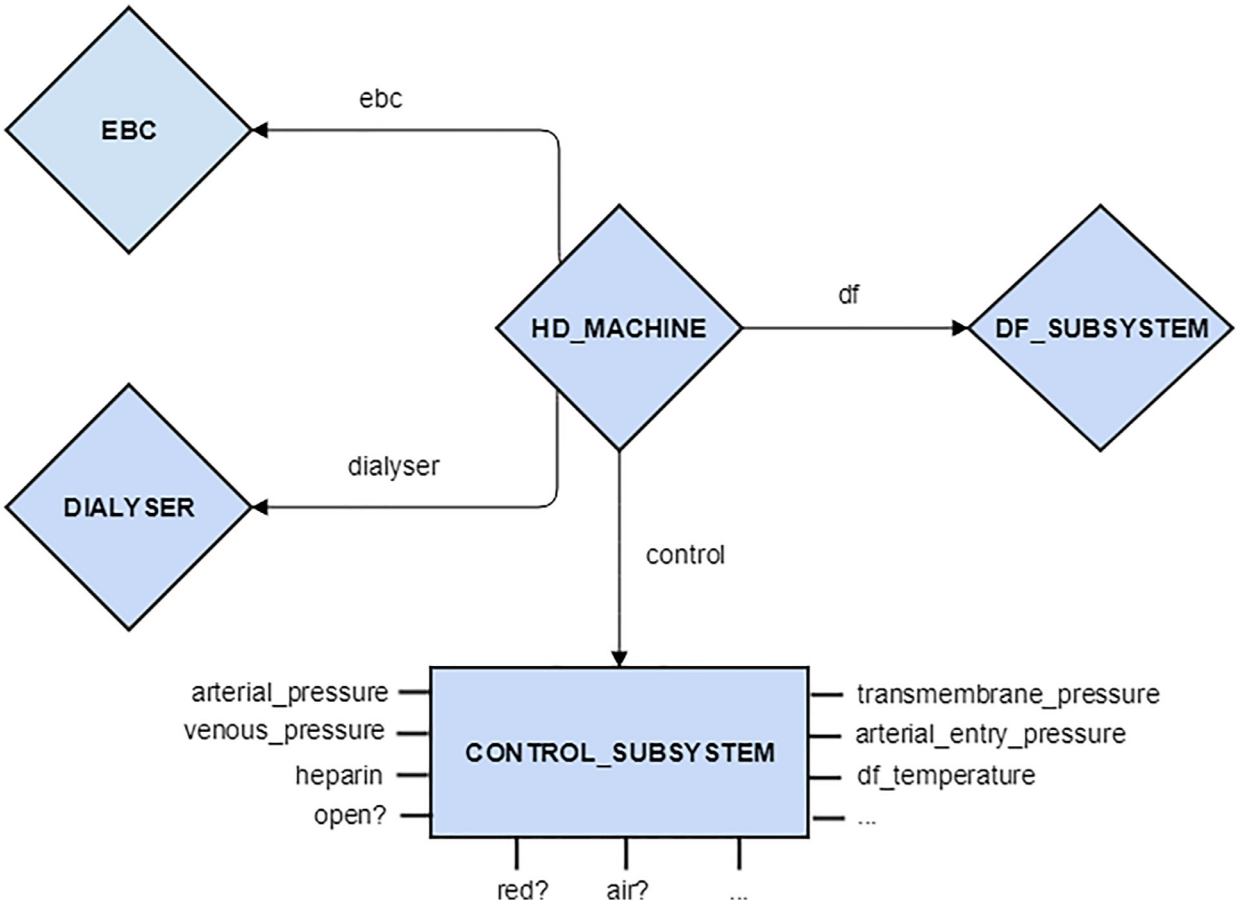

Fig. 6.2. Components of the hemodialysis machine (subdiagram).

transmembrane_pressure, heparin, open?, red?, air?, df_temperature}). (6.2)

The first four attributes all have the type (following Definition 3.1) $Pressure = REAL \to REAL$ capturing that pressure depends on time. The attribute heparin has the type $Volume = REAL \to REAL$ capturing that also the heparin volume can be manipulated over time. Fpr the attributes open?, red? and air? only two values are permitted, so the type is $YesNo = (\text{yes} : \mathbb{1}) \uplus (\text{no} : \mathbb{1})$ in all three cases. The attribute df_temperature capturing the desired temperature of the dialyser fluid has the type $Temperature = REAL \to REAL$. This will enable to control the temperature over time.

There are many additional attributes associated with the control unit such as all the rinsing parameters, dialyser fluid parameters and UF parameters defined in [35] that are set by the user and are used in the monitoring of HD machine. We omit a more detailed discussion of these parameters here.

The extracorporal blood circuit is a very important component. As illustrated in Fig. 6.3 the block type EBC of order 1 has six components, the elementary block types Arterial_Chamber, Venous_Chamber, Arterial_Pump, Heparin_Pump, Safety_Detector and Venous_Valve. Thus, the definition of the block type EBC (see Definition 3.3) takes the form

({ac : Arterial_Chamber, vc : Venous_Chamber, pp : Arterial_Pump,
sp : Heparin_Pump, safety : Safety_Detector, valve : Venous_Valve}, ∅). (6.3)

The arterial pump pumps the patient's blood into the arterial chamber, from where it flows into the dialyser for cleaning. In addition, the heparin pump pumps heparin into the blood to prevent coagulation. Cleaned blood flows from the dialyser into the venous chamber, from where it is returned to the patient's body. This return flow is monitored by the safety detector, which detects both blood and air. In the latter case the venous valve will be blocked.

The most important attributes of Arterial_Chamber are inflow, outflow and heparin, all of type *Volume*, and arterial_pressure of type *Pressure*. Likewise, Venous_Chamber has the attributes inflow and outflow of type *Volume*, and venous_pressure of type *Pressure*. The attributes of Arterial_Pump are inflow and outflow of type *Volume*, and arterial_entry_pressure of type *Pressure*, and Heparin_Pump has the attribute heparin of type *Volume*. Attributes of Safety_Detector are inblood and outblood of type *Volume* and air? and red? of type *YesNo*. Analogously, Venous_Valve has attributes inblood and outblood of type *Volume* and open? of type *YesNo*.

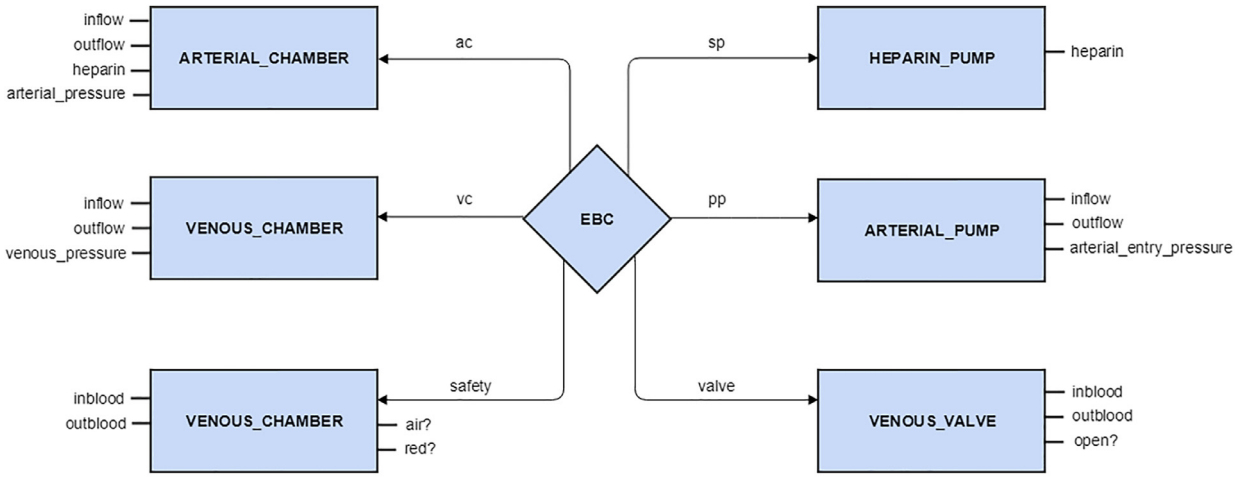

Fig. 6.3. Subdiagram for the extra corporal blood circuit.

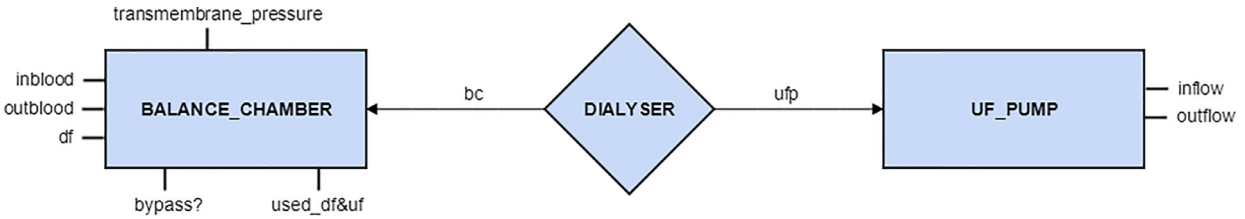

Fig. 6.4. Subdiagram for the dialyser component.

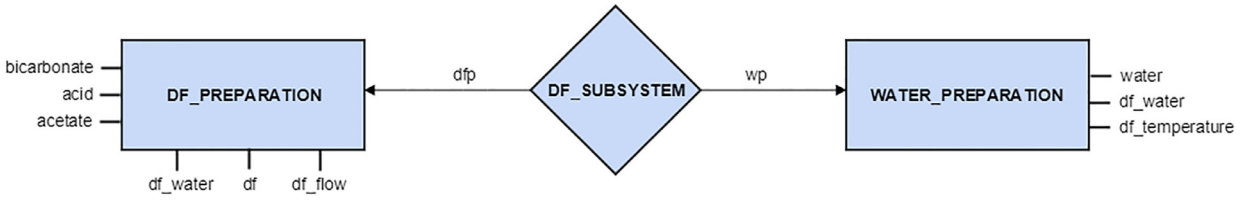

Fig. 6.5. Subdiagram for the dialyser fluid subsystem.

The structure of the dialyser component is illustrated in Fig. 6.4. A dialyser has a cylindric shape and is made of a semipermeable membrane configured as hollow fibers that separates blood from dialysate. It comprises the most important balance chamber and another pump for the ultra-filtrate. The balance chamber contains two chambers (one for blood and one for dialysate) allows each to be filled from one side while an identical volume is emptied to the other side. The total outlet fluid volume is always equal to the total input fluid volume. Each membrane has a magnetic sensor for monitoring purposes. Thus, the the definition of the block type DIALYSER takes the form

$$(\{\text{bc} : \text{BALANCE\_CHAMBER}, \text{ufp} : \text{UF\_PUMP}\}, \emptyset). \tag{6.4}$$

BALANCE_CHAMBER is an elementary component with attributes inblood, outblood, df and used_df&uf of type *Volume*, bypass? of type *YesNo*, and transmembrane_pressure of type *Pressure*. UF_PUMP is another elementary component with attributes inflow and outflow, both of type *Volume*.

The dialyzing fluid (DF) preparation component is responsible for concentrate preparation that consists of mixing the heated and degassed water with bicarbonate concentrate and acid concentrate. The accuracy of the DF concentration is controlled by the conductivity sensors. The DF water preparation component is responsible for degassing and heating the purified water, coming from the reverse osmosis system, to a predetermined temperature. The temperature is set by the user (usually 37°C), before the concentrate is prepared. The degassing chamber and the heater assembly are integral part of the system. Fig. 6.5 illustrates the subschema for the dialyser fluid subsystem, where the block type DF_SUBSYSTEM is defined as

$$(\{\text{wp} : \text{WATER\_PREPARATION}, \text{dfp} : \text{DF\_PREPARATION}\}, \emptyset). \tag{6.5}$$

The elementary block type WATER_PREPARATION has attributes water and df_water of type *Volume* and df_temperature of type *Temperature*. Attributes of the elemenary block type are df_water and df of type *Volume*, and df_flow, bicarbonate, acid and acetate of type *REAL*.

Note that all the definitions of block types above define a well-defined schema in the sense of Definition 3.4 with a diagram illustrated by Figs. 6.2–6.5. As the schema described and illustrated above is already rather refined, the principle of separation of concerns is already fulfilled, i.e. CONTROL_SUBSYSTEM is a discrete component, whereas all other elementary components are continuous.

6.2. Flow dependencies

Among the elementary components in the structural conceptual model above we have several flow dependencies. We denote these flow dependencies in the format `in` → `out` or `in-out` ↔ `in-out`, respectively. The former format identifies in in-port on the left-hand-side and an out-port on the right-hand-side, while the latter one identifies in-out-ports on both sides. In our model we basically have three types of flow dependencies:

1. A dependency may indicate a physical flow between two continuous components involving pliant attributes, usually with data types such as *Volume* or *Pressure*.
2. A dependency may indicate a flow of a measured sensor value from a continuous component to a discrete one.
3. A dependency may indicate a flow of an actuator value from a discrete component to a continuous one.

The first type of flow dependencies occurs with the blood flow, the flow of the dialysing fluid and the flow of the ultra-filtrate. Thus we have (following Definition 4.7)

ARTERIAL_PUMP.outflow → ARTERIAL_CHAMBER.inflow
HEPARIN_PUMP.heparin → ARTERIAL_CHAMBER.heparin
ARTERIAL_CHAMBER.outflow → BALANCE_CHAMBER.inblood
WATER_PREPARATION.df_water → DF_PREPARATION.df_water
DF_PREPARATION.df → BALANCE_CHAMBER.df
BALANCE_CHAMBER.used_df&uf → UF_PUMP.inflow
BALANCE_CHAMBER.outblood → VENOUS_CHAMBER.inflow
VENOUS_CHAMBER.outflow → SAFETY_DETECTOR.inblood
SAFETY_DETECTOR.outblood → VENOUS_VALVE.inblood (6.6)

In all these cases the flow dependency corresponds to a simple tube connection, so we can make assertions for all of them. We might argue that the dependencies are not simple equalities, the default for flow dependencies, but that there is a small time delay instead. We neglect these subtleties for now. We also neglect the possibility of broken tubes, for which additional sensors would be required measuring the pressure at the in- and out-ports, respectively, such that a control component could indicate an alarm in case the flow dependency is violated.

The most important sensor values in the extra-coporal blood circuit are the incoming blood flow and the arterial pressure associated with the arterial pump, the heparin volume associated with the heparin pump, the arterial and venous pressure in the respective chambers, and the signals air? and red? associated with the safety detector. These give rise to the following flow dependencies:

ARTERIAL_PUMP.inflow ↔ CONTROL_SUBSYSTEM.filling_bp_volume
ARTERIAL_PUMP.arterial_entry_pressure →
CONTROL_SUBSYSTEM.arterial_entry_pressure
HEPARIN_PUMP.heparin ↔ CONTROL_SUBSYSTEM.heparin
ARTERIAL_CHAMBER.arterial_pressure → CONTROL_SUBSYSTEM.arterial_pressure
VENOUS_CHAMBER.venous_pressure → CONTROL_SUBSYSTEM.venous_pressure
SAFETY_DETECTOR.air? → CONTROL_SUBSYSTEM.air?
SAFETY_DETECTOR.red? → CONTROL_SUBSYSTEM.red? (6.7)

Like the first of these dependencies there are more flow dependencies associated with rinsing, dialysing fluid and ultra-filtrate control parameters that we omitted in the model. These can be subject to further refinements. Also not that some of these flow dependencies use in-out ports, so they are bi-directional to ensure that there is also an associated actuator.

Similarly, transmembrane_pressure associated with the balance chamber, and df, df_water, df_temperature associated with the dialysis fluid subsystem represent important sensor data that are to flow to the control subsystem. This defines the following flow dependencies:

BALANCE_CHAMBER.transmembrane_pressure →

$$\begin{aligned}
&\text{Control_Subsystem.transmembrane_pressure}\\
&\text{DF_Preparation.df} \rightarrow \text{Control_Subsystem.df}\\
&\text{DF_Preparation.df_water} \rightarrow \text{Control_Subsystem.df_water}\\
&\text{Water_Preparation.df_temperature} \leftrightarrow \text{Control_Subsystem.df_temperature}
\end{aligned} \tag{6.8}$$

Finally, we have the following flow dependencies that are associated with actuators (not yet covered by the cases above):

$$\begin{aligned}
&\text{Control_Subsystem.open?} \rightarrow \text{Venous_Valve.open?}\\
&\text{Control_Subsystem.bypass?} \rightarrow \text{Balance_Chamber.bypass?}\\
&\text{Control_Subsystem.df_flow} \rightarrow \text{DF_Preparation.df_flow}
\end{aligned} \tag{6.9}$$

All these flow dependencies correspond to a simple cabling connections between sensors, actuators and the control unit, so we can make assertions for all of them.

6.3. Parametric constraints

For the proper functioning of the machine several parametric constraints (as Definition 4.8) must be satisfied. The following list contains a small selection of such constraints:

- When running the EBC with a 8/12 mm pump segment, the machine shall be able to perform a blood flow in the range of 30...600 ml/min with a tolerance of 10% (arterial pressure = −200 to +400 mmHg) and of 25% (arterial pressure = −300 to −200 mmHg). That is, we require

$$\mathcal{D}_t F(t) = f_{\text{blood}}(t) \Rightarrow 0.9\ bf \leq \frac{60(F(b) - F(a))}{b - a} \leq 1.1\ bf, \tag{6.10}$$

  where $a \leq b$ are any points in time, f_{blood} is one of the functions associated with Arterial_Pump.inflow or Arterial_Pump.outflow, and bf is the value in [30,600] set for the blood flow.
- During the therapy, if the blood pump is running and if the venous pressure in the venous chamber is lower than a lower alarm limit minus 10 mmHg (VPAbsoluteMin - 10 mmHg) for more than 5 s, then the control system shall stop the blood pump and execute an alarm signal. This requirement can be formalized as follows:

$$\forall a, b. b - a \geq 5 \Rightarrow \exists t. a \leq t \leq b \wedge vp(t) > \text{min\_pressure} - 10, \tag{6.11}$$

  where $vp(t)$ is the function associated with Venous_Chamber.venous_pressure.
- The common dependency between volume and pressure defines a parametric constraint between arterial_pressure and inflow on Arterial_Chamber:

$$\mathcal{D}_t\, ap(t) \;=\; \frac{-\,const}{inflow(t)^2}\, \mathcal{D}_t\, inflow(t) \tag{6.12}$$

  using functions $ap(t)$ and $inflow(t)$ associated with Arterial_Chamber. arterial_pressure and Arterial_Chamber.inflow, respectively.
- During the therapy, the control system shall monitor the blood flow in the EBC and if no flow is detected for more than 120 seconds, then the control system shall stop the blood pump and execute an alarm signal. This involves the parametric constraint

$$\mathcal{D}_t F(t) = f_{\text{blood}}(t) \Rightarrow (b - a \geq 120 \Rightarrow F(b) - F(a) > 0) \tag{6.13}$$

  using again arbitrary time points $a \leq b$ are any points in time and the function f_{blood} associated with Arterial_Pump.inflow.
- The liquids flowing in and out of the balance chamber of the dialyser must have the same volume, i.e.

$$\forall t.\ \text{inblood}(t) + \text{df}(t) = \text{outblood}(t) + \text{used\_df\&uf}(t) \tag{6.14}$$

  using the functions associated with the attributes Balance_Chamber.inblood, Balance_Chamber.outblood, Balance_Chamber.df and Balance_Chamber.used_df&uf.

Other dependencies as listed in [35] can be formalized analogously.

6.4. Event-B machines

In order to define the behavior of the hemodialysis machine we define several asynchronously operating Event-B machines. Accordingly, we will turn the block type Control_Subsystem into a block type of order 1 such that each Event-B machines will be associated with a unique elementary block type that appears as a component of Control_Subsystem. In addition, as discussed in Section 5 each Event-B machines is associated with a view on the schema.

Machine $\mathcal{M}_{safety\_air}$

There must exist a machine, name it $\mathcal{M}_{\text{safety\_air}}$, that is responsible for the safety of the patient in case air is detected in the venous blood stream. So, first define a new block type Safety_Air_Controller that inherits only the attributes air? and open? from Control_Subsystem. The simple function of an associated Event-B machine is to close the venous valve, i.e. set open? to "no", in case air is detected, i.e. the value of air? is "yes". Then the view $\mathcal{V}_{\text{safety\_air}}$ is defined by the following subschema:

HD_Machine = ({control : Control_Subsystem, ebc : EBC}, ∅)
Control_Subsystem = ({safety : Safety_Air_Controller}, ∅)
Safety_Air_Controller = (∅, {air?, open?})
EBC = ({safety : Safety_Detector, valve : Venous_Valve}, ∅)
Safety_Detector = (∅, {air?})
Venous_Valve = (∅, {outblood, open?})

We fix an identifier $i_{hd} \in ID$ and define three variables for the Event-B machine $\mathcal{M}_{\text{safety\_air}}$ as follows:

var_air = HD_Machine(i_{hd})!control!safety.air?
var_open = HD_Machine(i_{hd})!control!safety.open?
var_out = HD_Machine(i_{hd})!ebc!valve.air?

Due to the assertion for flow dependencies we also have the equalities

var_air = HD_Machine(i_{hd})!ebc!safety.air?
var_open = HD_Machine(i_{hd})!ebc!valve.open?

The associated types of these attributes in the structural conceptual model defines the following three invariants:

$$\text{var\_air} \in \mathit{YesNo}, \quad \text{var\_open} \in \mathit{YesNo}, \quad \text{var\_out} \in \mathit{Volume}$$

Note that due to these typing invariants the variables var_air and var_open are mode, while var_out is pliant.

The most important event in the machine $\mathcal{M}_{\text{safety\_air}}$ is *close_valve*, which is activated, when air is detected in the blood, and results in closing the venous valve. Accordingly, the out-streaming blood volume will become 0. This event is specified as follows:

Event *close_valve*
WHEN var_air = yes
THEN var_open := no var_out := $\mathbf{I}f.f(t) = 0$

Machine $\mathcal{M}_{control\_bp}$

Another machine is $\mathcal{M}_{\text{control\_bp}}$, which is used for controlling the arterial blood pump. As before we define another component Control_BP of Control_Subsystem, and the view $\mathcal{V}_{\text{control\_bp}}$ is defined by the following subschema:

HD_Machine = ({control : Control_Subsystem, ebc : EBC}, ∅)
Control_Subsystem = ({bp : Control_BP}, ∅)
Control_BP = (∅, {arterial_entry_pressure, ap_limit, bp_speed})
EBC = ({pp : Arterial_Pump}, ∅)
Arterial_Pump = (∅, {inflow, outflow, arterial_entry_pressure})

Using the same identifier $i_{hd} \in ID$ we define three variables for the Event-B machine $\mathcal{M}_{\text{control\_bp}}$ as follows:

var_in = HD_Machine(i_{hd})!ebc!pp.inflow
var_pressure = HD_Machine(i_{hd})!ebc!pp.arterial_entry_pressure
var_out = HD_Machine(i_{hd})!ebc!pp.outflow
bp_speed = HD_Machine(i_{hd})!control!bp.bp_speed
ap_limit = HD_Machine(i_{hd})!control!bp.ap_limit

The associated types of these attributes in the structural conceptual model defines the following three invariants:

$$\text{var\_in} \in \mathit{Volume}, \quad \text{var\_pressure} \in \mathit{Pressure}, \quad \text{var\_out} \in \mathit{Volume}$$

In addition, we require several local variables in the events:

- activate_bp and bp_status, both of type $\mathit{OnOff} = (\text{on} : \mathbb{1}) \uplus (\text{off} : \mathbb{1})$ are used as status variable for the activation of the pump and the status of the arterial blood pump, which may be on or off.

- accelerate_bp $\in$ *Accelerate* = (on : $\mathbb{1}$) $\uplus$ (accelerating : $\mathbb{1}$) $\uplus$ (off : $\mathbb{1}$) is used as status variable for the acceleration of the arterial blood pump, which may be on, off or accelerating.
- bp_timing and ap_timing, both of type *REAL* $\rightarrow$ *REAL* are linear, monotonically decreasing functions that are used for monitoring a set time interval. At the start of the interval the value of the function is the time to elapse, at the end the value is 0, i.e. when the value 0 is reached, the set time has elapsed. The two variables are used for the timing intervals for the blood pump activation and the pressure adjustment.

The following events are associated with the control of the arterial pressure, which is monitored by an automatically set limits window. The limits window is set 10 s after the last activation of the arterial pump. Thus we first require an event to activate the arterial pump:

EVENT *activate_blood_pump*
WHEN activate_bp = on
THEN bp_status := on
bp_timing := max(0, $\mathbf{I}f.\mathcal{D}_t f(t) = -1 \wedge f(now = 10)$)

The width and thresholds of the limits window are set by the control unit of the hemodialysis machine. The lower limit value is automatically adjusted during treatment. This means that the distance between the lower limit and the actual pressure decreases. This compensates for the hematocrit increase generally caused by the added ultra-filtrate. The adjustment is carried out every 5 minutes and adds up to 2.5 mmHg at a time. So we need an event to adjust the pressure limit:

EVENT *adjust_pressure_limit*
WHEN bp_status = on $\wedge$ ap_timing(now) = 0
THEN ap_limit := ap_limit + 2.5
ap_timing := max(0, $\mathbf{I}f.\mathcal{D}_t f(t) = -1 \wedge f(now) = 300$)

The minimum distance of 22.5 mmHg is always maintained. The lower pressure limit during hemodialysis is checked. An optimal interval is approximately 35 mmHg between the lower pressure limit and the current value. By changing the speed of the arterial blood pump for a brief period, it is possible to reposition the limits window. So we define events for increasing the pressure and accelerating the blood pump:

EVENT *increase_pressure*
WHEN var_pressure(now) - ap_limit $\leq$ 22.5
THEN accelerate_bp := on
EVENT *accelerate_blood_pump*
WHEN bp_status = on $\wedge$ accelerate_bp = on
THEN bp_speed := bp_speed + $const_1$
accelerate_bp := accelerating
var_in := $\mathbf{I}f.f(t) = \text{var\_in}(t) + const_2$
var_pressure := $\mathbf{I}f.\mathcal{D}_t f(t) = \dfrac{-const}{\text{var\_out}(t)^2}\,\mathcal{D}_t\,\text{var\_out}(t) \wedge$
$f(now) = \text{var\_pressure}(now)$)

Furthermore, the request that the blood pump is only accelerated for a brief period of time has to be taken care of by another event

EVENT *stop_accelerating_blood_pump*
WHEN accelerate_bp = accelerating $\wedge$
var_pressure(now) - ap_limit $\geq$ 35
THEN bp_speed := bp_speed - $const_1$
accelerate_bp := off

Other machines

The two machines above are examples for the capture of behavior in our model with direct link to Event-B. The important aspect is that these machines must run asynchronously. The are other machines, which we do not specify in this article. Clearly, the control subsystem must control the preparation of the dialyser fluid, which could be handled by a machine $\mathcal{M}_{\text{control\_df}}$. Furthermore, there are many safety alerts—actually, the vast majority of events in a hemodialysis machine is associated with safety monitoring (see the list of requirements in [35])—and all these safety alerts must be handled asynchronously. Here we dispense with the detailed development of corresponding Event-B machines.

7. Conclusion

In this article we developed a conceptual model for hybrid dynamic systems and exemplified its usage on the hemodialysis machine case study [35]. The conceptual model comprises a structural and a behavioral part. The former one is grounded in HERM [48] and emphasizes part-of-relationships among components and descriptive attributes associated with complex data types, in particular capturing continuous functions. Furthermore, complex structural constraints define conditions that must be satisfied in all states. Flow dependencies and parametric constraints allow us to capture continuous behavior by integrating differential equations.

The behavioral part in grounded in hybrid Event-B [9,10] and emphasizes multiple asynchronous machines with a common context and shared variables. It obeys restrictions arising from flow dependencies in the structural part which imply synchronized behavior. Therefore, individual machines with a synchronous behavior are associated with views on the structural model, which provides guidelines how to use the modeling language. Elementary components that are not further structured must not be hybrid; they can either by purely discrete or purely continuous. This separation principle defines another guideline.

The conceptual model has previously been investigated in the context of concurrent ASMs [18,21] and the landing gear case study [17]. Some of the ideas in the conceptual model that suggest extensions to Event-B are inspired by this research in the context of a related rigorous method.

The conceptual model encourages a systematic development method for hybrid systems, which first emphasizes the modeling aspect, i.e. the capture of components, their relationships and static and behavioral (parametric) constraints. This prevents developers from jumping too quickly into formalizing or even implementing tiny aspects of the behavior with the known danger to get lost in complexity for the overall development task. Looking into the various approaches addressing the hemodialysis case study [4,5,7,25,26,28,29] or the landing gear case study [6,8,30,34,45,47] this danger is real.

Furthermore, the conceptual model enforces the separation of concerns between continuous and discrete concepts. The latter ones lead to well-defined Event-B machines that are linked by views to the structural conceptual model as emphasized in this article. Concerning the continuous components the model enables to link them to descriptions in CAD systems and further to their production processes in CAM systems. In this way the behavior specification in the conceptual model will also be complemented for the physical components.

Event-B supports rigorous refinement, which has not yet been with in this article. The development of a refinement-based development method is left for future work. However, the perspective arising from the use of refinement is that our model does not only provide a method for high-level modeling with formal semantics, it also supports targeted systems development. Concerning the continuous extensions it may be necessary to integrate retrenchment [11], which is a particular form of refinement that takes care of the dependence on time in the continuous part of the specification. However, how retrenchment will enter the model is an open issue. Concerning asynchronous behavior, the notion of refinement has to be generalized to multiple Event-B machines. This defines the next steps of the research towards full support for hybrid systems engineering.

References

[1] Abrial J-R. Modeling in event-b - system and software engineering. Cambridge University Press; 2010.
[2] Abrial J-R, Butler M, Hallerstede S, Leuschel M, Schmalz M, Voisin L. Proposals for mathematical extensions for Event-B. http://deploy-eprints.ecs.soton.ac.uk/216/; 2010. Technical Report 216.
[3] Alur R, et al. The algorithmic analysis of hybrid systems. Theor Comput Sci 1995;138(1):3–34.
[4] Arcaini P, Bonfanti S, Gargantini A, Mashkoor A, Riccobene E. Integrating formal methods into medical software development: the ASM approach. Sci Comput Program 2018;158:148–67. doi:10.1016/j.scico.2017.07.003.
[5] Arcaini P, Bonfanti S, Gargantini A, Riccobene E. How to assure correctness and safety of medical software: the hemodialysis machine case study. In: Butler MJ, et al., editors. Proceedings of the abstract state machines, alloy, B, TLA, VDM, and Z (Proceedings ABZ 2016). Lecture Notes in Computer Science, Vol. 9675. Springer; 2016. p. 344–59.
[6] Arcaini P, Gargantini A, Riccobene E. Rigorous development process of a safety-critical system: from ASM models to Java code. STTT 2017;19(2):247–69. doi:10.1007/s10009-015-0394-x.
[7] Banach R. Hemodialysis machine in hybrid Event-B. In: Butler MJ, et al., editors. Proceedings of the abstract state machines, alloy, B, TLA, VDM, and Z (Proceedings ABZ 2016). Lecture Notes in Computer Science, Vol. 9675. Springer; 2016. p. 376–93.
[8] Banach R. The landing gear system in multi-machine hybrid Event-B. STTT 2017;19(2):205–28. doi:10.1007/s10009-015-0409-7.
[9] Banach R, Butler MJ, Qin S, Verma N, Zhu H. Core hybrid Event-B I: single hybrid Event-B machines. Sci Comput Program 2015;105:92–123.
[10] Banach R, Butler MJ, Qin S, Zhu H. Core hybrid Event-B II: multiple cooperating hybrid Event-B machines. Sci Comput Program 2017;139:1–35.
[11] Banach R, Jeske C. Retrenchment and refinement interworking: the tower theorems. Math Struct Comput Sci 2015;25(1):135–202.
[12] Banach R, Zhu H, Su W, Huang R. Continuous KAOS, ASM, and formal control system design across the continuous/discrete modeling interface: a simple train stopping application. Formal Asp Comput 2014;26(2):319–66.
[13] Banach R, Zhu H, Su W, Wu X. A continuous ASM modelling approach to pacemaker sensing. ACM Trans Softw Eng Methodol 2014;24(1) 2:1-2:40.
[14] Beierle C, Schewe K-D. Abstract State Machines with Exact Real Arithmetic. In: Butler Michael J, editor. Lecture Notes in Computer Science, 10817. Springer; 2018. p. 139–54. doi:10.1007/978-3-319-91271-4_10.
[15] Bjørner D. Software engineering 3 – domains, requirements, and software design. Texts in Theoretical Computer Science. An EATCS Series. Springer; 2006. ISBN 978-3-540-21151-8. doi:10.1007/3-540-33653-2.
[16] Bjørner D. Domain engineering - technology management, research and engineering. COE Research Monograph Series, Vol. 4. JAIST; 2009. ISBN 978-4-903092-17-1.
[17] Boniol F, Wiels V. The landing gear system case study. In: Boniol F, Wiels V, Aït Ameur Y, Schewe K-D, editors. ABZ 2014: the landing gear case study. Communications in Computer and Information Science, Vol. 433. Springer; 2014. p. 1–18.
[18] Börger E, Schewe K-D. Concurrent Abstract State Machines. Acta Inform 2016;53(5):469–92.
[19] Börger E, Stärk R. Abstract state machines. Berlin Heidelberg New York: Springer-Verlag; 2003.
[20] Buga A, Mashkoor A, Nemeş ST, Schewe K-D, Songprasop P. Conceptual modelling of hybrid systems - structure and behaviour. In: Ouhammou Y, et al., editors. Proceedings of the 7th international conference model and data engineering (MEDI 2017). Lecture Notes in Computer Science, Vol. 10563. Springer; 2017. p. 277–90.
[21] Buga A, Nemeş ST, Schewe K-D, Songprasop P. A conceptual model for systems engineering and its formal foundation. In: Sornlertlamvanich V, et al., editors. Proceedings of the 27th international conference on information modelling and knowledge bases (EJC 2017). Frontiers in Artificial Intelligence and Applications, Vol. 301. IOS Press; 2018. p. 1–20.
[22] Dupont G. Event-B models – verbatim version. Tech. Rep.. IRIT; 2018.
[23] Dupont G, Aït-Ameur Y, Pantel M, Singh NK. Proof-based approach to hybrid systems development: Dynamic logic and Event-B. In: Butler MJ, et al., editors. Proceedings of the abstract state machines, alloy, B, TLA, VDM, and Z (ABZ 2018). Lecture Notes in Computer Science, Vol. 10817; 2018. p. 155–70. doi:10.1007/978-3-319-91271-4_11.
[24] Enterprise Architect – UML design tools and UML case tools for software development. http://www.sparxsystems.com.au/products/ea/index.html; 2016.

[25] Fayolle T, Frappier M, Gervais F, Laleau R. Modelling a hemodialysis machine using algebraic state-transition diagrams and B-like methods. In: Butler MJ, et al., editors. Proceedings of the abstract state machines, alloy, B, TLA, VDM, and Z (Proceedings ABZ 2016). Lecture Notes in Computer Science, Vol. 9675. Springer; 2016. p. 394–408.
[26] Gomes AO, Butterfield A. Modelling the haemodialysis machine with Circus. In: Butler MJ, et al., editors. Proceedings of the abstract state machines, alloy, B, TLA, VDM, and Z (Proceedings ABZ 2016). Lecture Notes in Computer Science, Vol. 9675. Springer; 2016. p. 409–24.
[27] Gurevich Y. Sequential Abstract State Machines capture sequential algorithms. ACM Trans Comput Logic 2000;1(1):77–111.
[28] Hoang TS, Snook CF, Fathabadi AS, Butler MJ, Ladenberger L. Validating and verifying the requirements and design of a haemodialysis machine using the Rodin toolset. Sci Comput Program 2018;158:122–47. doi:10.1016/j.scico.2017.11.002.
[29] Hoang TS, Snook CF, Ladenberger L, Butler MJ. Validating the requirements and design of a hemodialysis machine using iUML-B, BMotion Studio, and Co-simulation. In: Butler MJ, et al., editors. Proceedings of the abstract state machines, alloy, B, TLA, VDM, and Z (Proceedings ABZ 2016). Lecture Notes in Computer Science, Vol. 9675. Springer; 2016. p. 360–75.
[30] Ladenberger L, Hansen D, Wiegard H, Bendisposto J, Leuschel M. Validation of the ABZ landing gear system using Pro-B. STTT 2017;19(2):187–203. doi:10.1007/s10009-015-0395-9.
[31] Lamport L. Hybrid systems in TLA $^{+}$. In: Grossman RL, et al., editors. Hybrid systems. LNCS, Vol. 736. Springer; 1992. p. 77–102.
[32] Leuschel M, Börger E. A compact encoding of sequential asms in event-b. In: Butler MJ, et al., editors. Proceedings of the abstract state machines, alloy, B, TLA, VDM, and Z (ABZ 2016). LNCS, Vol. 9675. Springer; 2016. p. 119–34.
[33] Magicdraw. http://www.nomagic.com/products/magicdraw.html; 2016.
[34] Mammar A, Laleau R. Modeling a landing gear system in Event-B. STTT 2017;19(2):167–86. doi:10.1007/s10009-015-0391-0.
[35] Mashkoor A. The hemodialysis machine case study. In: Butler MJ, et al., editors. Proceedings of the abstract state machines, alloy, B, TLA, VDM, and Z (Proceedings ABZ 2016). Lecture Notes in Computer Science, Vol. 9675. Springer; 2016. p. 329–43.
[36] OMG Systems Modeling Language (OMG SysML), Version 1.4. 2015. OMG document number formal/2015-06-03; URL http://www.omg.org/spec/SysML/1.4/.
[37] Platzer A. Analog and hybrid computation: dynamical systems and programming languages. Bull EATCS 2014;114:152–99.
[38] PTC Integrity Modeler. http://www.ptc.com/model-based-systems-engineering/integrity-modeler; 2016.
[39] Sarstedt S. Semantic foundation and tool support for model-driven development with UML 2 activity diagrams. Universität Ulm; 2006.
[40] Schewe K-D. On the unification of query algebras and their extension to rational tree structures. In: Orlowska ME, Roddick JF, editors. Proceedings of the twelfth Australasian database conference (ADC 2001). Bond University, Queensland, Australia: IEEE Computer Society; 2001. p. 52–9.
[41] Schewe K-D. UML: a modern dinosaur? A critical analysis of the unified modelling language. In: Jaakkola H, Kangassalo H, Kawaguchi E, editors. Information modelling and knowledge bases XII. Frontiers in Artificial Intelligence and Applications, Vol. 67. IOS Press; 2001. p. 185–202.
[42] Schewe K-D Extensions to Event-B to support concurrency in cyber-physical systems. In: Méry D, et al., editors. Proceedings of the 8th international conference model and data engineering (MEDI 2018). LNCS. Springer; 2018. To appear.
[43] Songprasop P. Structural modelling in systems engineering. Master's thesis. Johannes-Kepler-University Linz; 2017.
[44] Stan-Ober I. Harmonisation des languages de modelisation avec des extensions orientées-object et une sémantique exécutable. PhD thesis. Institute National Polytechnique de Toulouse; 2001.
[45] Su W, Abrial J-R. Aircraft landing gear system: approaches with Event-B to the modeling of an industrial system. STTT 2017;19(2):141–66. doi:10.1007/s10009-015-0400-3.
[46] Su W, Abrial J-R, Zhu H. Formalizing hybrid systems with event-b and the rodin platform. Sci Comput Program 2014;94:164–202.
[47] Teodorov C, Dhaussy P, Leroux L. Environment-driven reachability for timed systems - safety verification of an aircraft landing gear system. STTT 2017;19(2):229–45. doi:10.1007/s10009-015-0401-2.
[48] Thalheim B. Entity-relationship modeling – foundations of database technology. Springer; 2000.