
Proceedings Paper:
Lefticaru, R., Bakir, M.E., Konur, S. et al. (2018) Modelling and validating an engineering application in kernel P systems. In: Gheorghe, M., Rozenberg, G., Salomaa, A. and Zandron, C., (eds.) Membrane Computing. CMC 2017. 18th International Conference, CMC 2017, 25-28 Jul 2017, Bradford, UK. Lecture Notes in Computer Science, 10725 . Springer International Publishing, pp. 183-195. ISBN: 9783319733586. ISSN: 0302-9743. EISSN: 1611-3349.

https://doi.org/10.1007/978-3-319-73359-3_12

Modelling and Validating an Engineering Application in Kernel P Systems

Raluca Lefticaru[1,2], Mehmet Emin Bakir[3], Savas Konur[1], Mike Stannett[3], Florentin Ipate[2]

[1] School of Electrical Engineering and Computer Science, University of Bradford, West Yorkshire, Bradford BD7 1DP, UK
{r.lefticaru,s.konur}@bradford.ac.uk
[2] Department of Computer Science, University of Bucharest, Str. Academiei nr. 14, 010014, Bucharest, Romania
florentin.ipate@ifsoft.ro
[3] Department of Computer Science, The University of Sheffield, Regent Court, 211 Portobello, Sheffield S1 4DP, UK
{mebakir1,m.stannett}@sheffield.ac.uk

Abstract. This paper illustrates how kernel P systems (*kP systems*) can be used for modelling and validating an engineering application, in this case a cruise control system of an electric bike. The validity of the system is demonstrated via formal verification, carried out using the kPWorkbench tool. Furthermore, we show how the kernel P system model can be tested using automata and X-machine based techniques.
Keywords: membrane computing; kernel P systems; cruise control; electric bike; bicycle; verification; testing.

1 Introduction

Nature inspired computational approaches have been the focus of research for several decades. Membrane computing [21] is one of these paradigms that has recently been through significant developments and achievements. For the most up to date results, we refer the reader to [22]. The main computational models are called *P systems*, inspired by the functioning and structure of the living cells.

In recent years, various types or classes of P systems have been introduced and applied to different problems. While these variants provide more flexibility in modelling, this has inevitably resulted in a large pool of P system variants, which do not have a coherent integrating view.

Kernel P (kP) systems have been introduced to unify many variants of P system models, and combine a blend of various P system features and concepts, including (i) complex guards attached to rules, (ii) flexible ways to specify the system structure and dynamically change it and (iii) various execution strategies for rules and compartments.

Kernel P systems are supported by a software suite, called kPWorkbench [5]. The platform integrates several tools to simulate and verify kP systems models written in a modelling language, called *kP-Lingua*, capable of mapping the kernel P system specification into a machine readable representation.

The usability and efficiency of kP systems have been illustrated by a number of representative case studies, ranging from systems and synthetic biology, e.g. quorum sensing [18], genetic Boolean gates [23] and synthetic pulse generators [1], to some classical computational problems, e.g. sorting [6], broadcasting [10] and subset sum [5].

Here, as an engineering application, we focus on an *e-bike cruise control system*. An *e-bike* (electric bicycle) is a bicycle that uses an integrated rechargeable battery and an electric motor, which provides propulsion. A *cruise control* is an advanced driver-assistance system technology that automatically regulates the speed of a transportation system (such as motor vehicle or electric bicycle) set by the user. From a system design perspective, the validation of the operational safety of any component/feature is very crucial [24].

In this paper, we will model an e-bike cruise control system using kernel P systems and verify its behaviour using the kPWorkbench verification environment. We also show how the kernel P system model can be tested using automata and X-machine based techniques.

This paper is structured as follows: Section 2 introduces the preliminaries and theoretical background. Section 3 presents our modelling approach using kernel P systems, while Sections 4 and 5 present the verification and testing approaches. Finally, conclusions and further work are presented in Section 6.

2 Background

This section briefly presents the cruise control system, then gives the basic definitions regarding kernel P systems [9], a presentation of the kPWorkbench software suite, and previous testing approaches for membrane systems.

2.1 Cruise control system for an electric bicycle

In this paper, we focus on an e-bike cruise control system. By controlling the speed of the e-bike (or other transportation system), this feature makes the driving experience easier as the user does not have to use the accelerator or brake. For an e-bike system a cruise control feature also assists the user by improving the control of the journey time and controlling the level of exercise undertaken.

From a system design point of view, however, adding a new feature brings in new challenges for the operational safety of the new functionality [24]. Thus validation of additional functionalities of any new technology and their impact on other components of an existing system is important. This will be our focus in this paper. The behaviour of the e-bike cruise control considered in this paper is shown in Figure 1.

In a previous paper [20], a similar e-bike case study has been used to illustrate an integrated approach, combining software engineering methodologies (verification and model-based testing) with notations and methods from system engineering. Although the two state machines corresponding to the e-bike

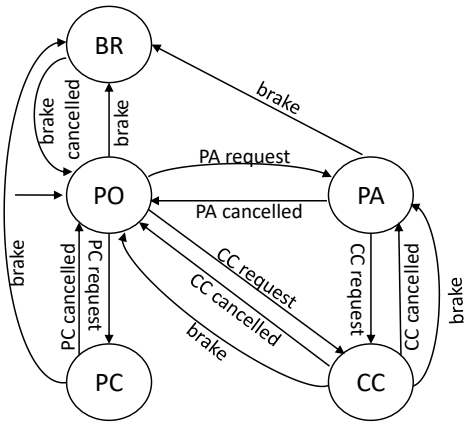

Fig. 1. The state machine representing the behaviour of the e-bike cruise control system considered in this paper. The system works as follows:

- At any time, the system can be at any of the following states:
  (i) pedal bike (Pedal Only – PO, for short)
  (ii) pedal bike with power assistance (Pedal Assist – PA)
  (iii) maintain constant speed (Cruise Control – CC)
  (iv) pedal to charge battery (Pedal Charge – PC)
  (v) brake (Brake – BR).
- CC can be activated from PO or PA.
- If CC is cancelled, the system returns to the state from where it was activated i.e., PO or PA, respectively.
- Pedal assist can be requested when the user is pedalling.
- Pedal charge can be requested when the user is pedalling.
- When the user brakes from CC mode, the system returns to PA/PO before going to BR (if brake is still held).
- BR can be reached from PO, PA or PC.
- If the user releases the brake, the system goes to PO, no matter which was the state before Brake. This happens because from the Brake state, after releasing the brake lever, one can only start to pedal and enter in PO mode. In order to enter in PA mode, the user must first start to pedal and then make a pedal assist request.

system (from current paper and from [20]) have many similarities, the current approach adopts kernel P systems as modelling formalism and further illustrates how these can be used for simulating and validating an engineering application.

2.2 Kernel P systems

We first begin recalling the formal definition of kernel P systems (or kP systems).

Definition 1. *A* kP system *of degree n is a tuple $k\Pi = (A, \mu, C_1, \ldots, C_n, i_0)$, where*

- *A is a finite set of elements called objects;*

- μ *defines the membrane structure, which is a graph,* (V, E)*, where* V *is a set of vertices representing components (compartments), and* E *is a set of edges, i. e., links between components;*
- $C_i = (t_i, w_{i,0})$*,* $1 \leq i \leq n$*, is a compartment of the system consisting of a* compartment type*,* t_i*, from a set* T *and an initial multiset,* $w_{i,0}$ *over* A*; the type* $t_i = (R_i, \rho_i)$ *consists of a set of evolution rules,* R_i*, and an execution strategy,* ρ_i*;*
- i_0 *is the output compartment where the result is obtained.*

Kernel P systems have features inspired by object-oriented programming: one *compartment type* can have one or more *instances*. These instances share the same set of rules and execution strategies (so will deliver the same functionality), but they may contain different multisets of objects and different neighbours according to the graph relation specified by (V, E). Within the kP systems framework, the following types of evolution rules have been considered so far:

- *rewriting and communication* rule: $x \longrightarrow y\{g\}$, where $x \in A^+$ and y represents a multiset of objects over A^* with potential different compartment type targets (each symbol from the right side of the rule can be sent to a different compartment, specified by its type; if multiple compartments of the same type are linked to the current compartment, then one is randomly chosen to be the target). Unlike cell-like P systems, the targets in kP systems indicate only the types of compartments to which the objects will be sent, not particular instances. Also, for kP systems, complex *guards* can be represented, using multisets over A with relational and Boolean operators [9].
- *structure changing* rules: membrane division, membrane dissolution, link creation and link destruction rules, which all may also incorporate complex guards and that are covered in detail in [9].

In addition to its evolution rules, each compartment type in a kP system has an associated *execution strategy*. The rules corresponding to a compartment can be grouped in blocks, each having one of the following strategies:

- *sequential*: if the current rule is applicable, then it is executed, advancing towards the next rule/block of rules; otherwise, the execution terminates;
- *choice*: a non-deterministic choice within a set of rules. One and only one applicable rule will be executed if such a rule exists, otherwise the whole block is simply skipped;
- *arbitrary*: the rules from the block can be executed zero or more times by non-deterministically choosing any of the applicable rules;
- *maximal parallel*: the classic execution mode used in membrane computing.

These execution strategies and the fact that in any compartment several blocks with different strategies can be composed and executed offer a lot of flexibility to the kP system designer, similarly to procedural programming.

2.3 kPWorkbench

Kernel P systems are supported by an integrated software suite, kPWorkbench [5], which employs a set of simulation and formal verification tools and methods that permit simulating and verifying kP system models, written in kP-Lingua.

The verification component of kPWorkbench checks the correctness of kP system models by exhaustively analysing all possible behaviours. In order to facilitate the specification of system requirements, kPWorkbench features a property language, called *kP-Queries*, which comprises a list of property patterns written as natural language statements. The properties expressed in *kP-Queries* are verified using the Spin [13] and NuSMV [3] model checkers after being translated into corresponding *Linear Temporal Logic (LTL)* and *Computation Tree Logic (CTL)* syntax.

The simulation component features a native simulator [2, 19], which allows the users to simulate kP system models efficiently. In addition, kPWorkbench integrates the Flame simulator [4, 23], a general purpose large scale agent based simulation environment, based on a method that allows users to express kP systems as a set of communicating X-machines [11].

2.4 Kernel P systems testing

When testing a kP system model, an automata model needs to be constructed first, based on the computation tree of the kP system. As, in general, the computation tree may be infinite and cannot be modelled by a finite automaton, an approximation of the tree is used. This approximation is obtained by limiting the length of any computation to an upper bound k and considering only computations up to k transitions in length. This approximation is then used to construct a deterministic finite cover automaton (DFCA) of the model [6–8].

However, in the case of the e-bike, this can be naturally modelled by a state-based formalism and, furthermore, the kP system was derived from such a model (Fig. 1). Therefore one can use this state-based model in testing. It can be observed, however, that the model is not exactly a finite automaton since an additional variable is used to decide to which state (PO or PA) the e-bike returns when the Cruise Control facility is cancelled[1]. Such a formalism, that combines a finite state machine like control with data structures is the *stream X-machine* [12].

A stream X-machine (SXM) is like a finite automaton in which the transitions are labelled by partial functions (or, more generally, relations) instead of mere symbols. Formally,

[1] One could build a Finite State Automaton with two extra states (CCPO and CCPA, that allow to come back to PO and PA, respectively, when CC facility is cancelled), plus other necessary transitions from/to these states, in order to simulate the same behaviour of the e-bike model. However, the corresponding X-machine model, having one memory variable instead of the 2 extra states, has the advantage of keeping the control structure simpler; having less states it's easier to be read and the states correspond exactly to the device modes.

Definition 2. *A* stream X-Machine *(abbreviated* SXM*) is a tuple*

$$Z = (\Sigma, \Gamma, Q, M, \Phi, F, q_0, m_0),$$

where:

- Σ *is the finite* input alphabet.
- Γ *is the finite* output alphabet.
- Q *is the finite set of* states.
- M *is a (possibly infinite) set called* memory.
- Φ *is a finite set of distinct* processing functions*; a processing function is a non-empty (partial) function of type* $M \times \Sigma \longrightarrow \Gamma \times M$.
- F *is the (partial)* next-state function*,* $F : Q \times \Phi \longrightarrow Q$.
- $q_0 \in Q$ *is the initial state.*
- $m_0 \in M$ *is the initial memory value.*

Intuitively, an SXM Z can be thought as a finite automaton with the arcs labelled by functions from the set Φ. The automaton $A_Z = (\Phi, Q, F, q_0)$ over the alphabet Φ is called *the associated finite automaton* (abbreviated *associated FA*) of Z and is usually described by a state-transition diagram. As with any automaton, the function F may be extended to take sequences from Φ^*, to form the function $F^* : Q \times \Phi^* \longrightarrow Q$. We will write $L_{A_Z}(q) = \{p \in \Phi^* \mid (q, p) \in dom\ F^*\}$ to denote the set of paths that can be traced out of state q. When $q = q_0$, this will be called the *language accepted* by Z and denoted L_{A_Z}.

3 kP model for e-bike cruise control

In [20], the e-bike cruise control system has been manually coded into different formal models for verification and model-based testing, which is a very challenging and time consuming process. Also, any change in the system model requires the modification of all formal models. This issue, the direct coding from the system description, has been highlighted in various engineering applications, e.g. real-time systems [15], safety critical systems [16], pervasive systems [17].

Using kernel P systems as modelling language provides some practical advantages. Namely, several verification and simulation methods integrated into kPWorkbench are readily available; hence several complementary analysis can be performed, which allows more in-depth analysis of the system. Since kPWorkbench automatically translates a kP system model written in kP-Lingua into the corresponding formal syntax, users do not need to carry out manual encoding to access the tools. In addition, kP-Lingua has a simple language, which makes it much easier to express system models.

In this section we present a kP system model for the cruise control system described as a state machine in Fig. 1. The corresponding kP system has two compartment types: (1) $tEvent$, in charge of generating all possible events (or inputs from the user) and sending them to $tEBike$; (2) $tEBike$, receiving these events and processing them according to state machine rules. The $tEBike$ will

always contain only one element of the set $\{PO, CC, PC, PA, BR\}$ representing the current state of the machine, and might have other elements such as $\{pa, pc, cc, br, pac, pcc, ccc, brc\}$ representing the event received from $tEvent$ or $\{po2cc, pa2cc\}$ as objects recording which was the previous state before CC. The event names are lower case always, compared to their upper case states counterparts, e.g. pa, cc for pedal assist, cruise control request, while brc, pcc represent brake cancelled or pedal charge cancelled.

Figure 2 presents the kP-Lingua source code corresponding to our model of e-bike cruise control. The execution strategy is *choice* for both compartment types, but in this particular case the maximal parallelism strategy would have provided the same functionality. The computation is infinite and due to the nondeterminism of the model we would like to check if some properties hold for any possible computation. The kP-Lingua model and verification files discussed here are available for download on the kPWorkbench website[2].

4 Verification

In this section, we check various properties of the e-bike model to verify that the model satisfies the system requirements using the verification component of kPWorkbench. The tool translates the kP-Lingua model of the e-bike system into the NuSMV modelling language. Similarly, the properties written in kP-Queries (using natural language statements) are translated into the NuSMV property specification language (the translation can be in LTL or CTL).

Table 1 shows the verification results of the e-bike model properties. The first column shows the property id; the second column describes the properties informally; the third column shows the formal properties expressed in kP-Queries (which are then translated into LTL and CTL in NuSMV syntax); and the last column illustrates the verification result.

The first property checks whether BR is reachable from any state after brake requested. The property holds because BR can be activated directly from PO, PA and PC, and there are paths from CC to BR, too, over PO and PA. The second property verifies that after BR is activated, the system will either stay in BR or move to PO. As expected, this property also holds, because BR cannot request any states other than itself or PO. The properties from 3 to 8 test different transitions from/to the CC state. For example, properties 4 and 5 verify that after CC is cancelled, the system will return to the state from which it was activated, i.e., PO or PA. Properties 3–8 all hold, except for property 8, which is false. This property checks the existence of states (other than PO and PA) from which we may have direct access to the CC state. However, only PO and PA can access CC, so the property does not hold. The remaining properties, 9–12, check the existence/absence of transitions from/to PC. Again, all properties hold except property 11. This property asserts that PC can be activated from a state other than PO, whereas in fact only PO can activate PC. Therefore, it does

[2] http://kpworkbench.org/index.php/case-studies

```
type tEvent{
 choice{
  g -> g, br(tEBike).
  g -> g, cc(tEBike).
  g -> g, pa(tEBike).
  g -> g, pc(tEBike).
  g -> g, brc(tEBike).
  g -> g, ccc(tEBike).
  g -> g, pac(tEBike).
  g -> g, pcc(tEBike).
  }
}

type tEBike{
 choice{
  PO, br -> BR.
  PO, cc -> CC, po2cc.
  PO, pa -> PA.
  PO, pc -> PC.
  PO, brc -> PO.
  PO, ccc -> PO.
  PO, pac -> PO.
  PO, pcc -> PO.

  PA, br -> BR.
  PA, cc -> CC, pa2cc.
  PA, pa -> PA.
  PA, pc -> PA.
  PA, brc -> PA.
  PA, ccc -> PA.
  PA, pac -> PO.
  PA, pcc -> PA.

  PC, br -> BR.
  PC, cc -> PC.
  PC, pa -> PC.
  PC, pc -> PC.
  PC, brc -> PC.
  PC, ccc -> PC.
  PC, pac -> PC.
  PC, pcc -> PO.

  CC, br, pa2cc -> PA.
  CC, br, po2cc -> PO.
  CC, cc -> CC.
  CC, pa -> CC.
  CC, pc -> CC.
  CC, brc -> CC.
  CC, ccc, po2cc -> PO.
  CC, ccc, pa2cc -> PA.
  CC, pac -> CC.
  CC, pcc -> CC.

  BR, br -> BR.
  BR, cc -> BR.
  BR, pa -> BR.
  BR, pc -> BR.
  BR, brc -> PO.
  BR, ccc -> BR.
  BR, pac -> BR.
  BR, pcc -> BR.
 }
}
cEvent {g} (tEvent).
cEBike {PO} (tEBike).
cEvent - cEBike.
```

Fig. 2. kP-Lingua code for the e-bike cruise control system

not hold. The verified properties validate that the e-bike system works as desired.

5 Testing

In this section we show how the kernel P system model from Section 3 can be tested using automata and X-machine based techniques.

For the kP system described in Section 3, the associated stream X-machine (SXM) will be defined as follows:

Table 1. Verified properties

#	Description	kP-Queries	Res.
1	Whenever brake is requested, it will eventually be activated.	CTL: cEBike.br >0 **followed-by** cEBike.BR >0	T
2	BR either stays same or it can activate only PO	LTL: **always** ((cEBike.BR >0) **implies** (**next** (cEBike.PO >0 **or** cEBike.BR >0)))	T
3	The user should be able to request / activate CC only from PO or PA	LTL: **never** ((cEBike.BR >0 **or** cEBike.PC >0) **and** (**next** (cEBike.CC >0)))	T
4	If CC activated from PO, then the system will return to PO after CC cancel or brake request	LTL: **always** ((cEBike.CC >0 **and** cEBike.po2cc >0) **and** (cEBike.ccc >0 **or** cEBike.br >0) **implies** (**next**(cEBike.PO >0)))	T
5	If CC activated from PA, then the system will return to PA after CC cancel or brake request	LTL: **always** ((cEBike.CC >0 **and** cEBike.pa2cc >0) **and** (cEBike.ccc >0 **or** cEBike.br >0) **implies** (**next**(cEBike.PA >0)))	T
6	When brake is requested in CC the system returns to PA or PO	LTL: **always** ((cEBike.CC >0 **and** cEBike.br >0) **implies** (cEBike.BR >0 **preceded-by** (cEBike.PO >0 **or** cEBike.PA >0)))	T
7	The system should not transit directly from CC to Brake directly	LTL: **never** ((cEBike.CC >0 **and** cEBike.br >0) **and** (**next** (cEBike.BR >0)))	T
8	CC can be activated from a state other than PO or PA	CTL: (**not** (cEBike.PO >0 **or** cEBike.PA >0)) **until** cEBike.CC >0	F
9	PA and PC cannot directly activate each other	LTL: **never** ((cEBike.PA >0 **and** (**next** (cEBike.PC >0))) **or** (cEBike.PC >0 **and** (**next** (cEBike.PO >0))))	T
10	CC and PC cannot directly activate each other	LTL: **never** ((cEBike.CC >0 **and** (**next** (cEBike.PC >0))) **or** (cEBike.PC >0 **and** (**next** (cEBike.CC >0))))	T
11	PC can be activated from a state other than PO	CTL: (**not** (cEBike.PO >0)) **until** cEBike.PC >0	F
12	PC can activate PC, PO or BR only	LTL: **always** (cEBike.PC >0 **implies** (**next** ((((cEBike.PC >0) **or** (cEBike.PO >0)) **or** (cEBike.BR >0)))))	T

- the set of states is $Q = \{PO, PA, PC, CC, BR\}$;
- the set of inputs is $\Sigma = \{pa, cc, pc, br, brc, pac, ccc, pcc\}$
- there are no explicit outputs; in order to make the transition observable we consider the output to be the next state for each transition, so the set of outputs is the same as the set of states, $\Gamma = Q$.

- the memory is $M = \{m\}$, $m \in \{\lambda, pa2cc, po2cc\}$ (one memory variable m, where λ represents an undefined value, and $pa2cc, po2cc$ are used to record the last state before enabling the CC feature);
- each processing function is determined by a rewriting rule in $tEBike$, e.g., the $PO, pa \rightarrow PA$ rule induces the processing function $\phi_{PO,pa,PA}$ defined by $\phi_{PO,pa,PA}(m, pa) = (PA, m)$, $m \in M$; the $PO, cc \rightarrow CC, po2cc$ rule induces processing function $\phi_{PO,cc,CC}(m, cc) = (CC, po2cc)$, $m \in M$; the $CC, ccc, po2cc \rightarrow PO$ rule induces processing function $\phi_{CC,ccc,PO}(po2cc, ccc) = (PO, \lambda)$;
- the next-state function is defined by $F(q, \phi_{q,\sigma,q'}) = q'$ for every $q, q' \in Q$, $\sigma \in \Sigma$ such that $\phi_{q,\sigma,q'} \in \Phi$;
- the initial state is $q_0 = PO$;
- the initial memory is $m_0 = \lambda$.

Now, suppose we want to test an implementation of a system specified as an SXM. The testing techniques presented in [12, 14] generate test suites that guarantee that the implementation conforms to the model, provided that some *design for test* conditions are satisfied and the tester is able to estimate the maximum number of states the implementation may have. We denote by β the difference between this estimated upper bound on the number of states of the implementation under test and the number of states of the model.

In order to generate a test suite from a SXM, two set of paths from the associated automaton will have to be constructed: a state cover and a characterisation set.

A *transition cover* of a SXM Z is a set $S \subseteq \Phi^*$ such that for every state $q \in Q$ of Z there is $p \in S$ such that p reaches state q, i.e. $F^*(q_0, p) = q$. In our example, the empty sequence λ reaches the initial state PO, $\phi_{PO,pa,PA}$ reaches PA, $\phi_{PO,pc,PC}$ reaches PC, $\phi_{PO,cc,CC}$ reaches CC and $\phi_{PO,br,BR}$ reaches BR, thus $S = \{\lambda, \phi_{PO,pa,PA}, \phi_{PO,pc,PC}, \phi_{PO,cc,CC}, \phi_{PO,br,BR}\}$ is a state cover of Z.

A *characterization set* of a SXM Z is a set $W \subseteq \Phi^*$ such that for every two distinct states $q, q' \in Q$ there is $p \in W$ such that p distinguishes between q and q', i.e $F^*(q, p)$ is defined and $F^*(q', p)$ is not defined or $F^*(q, p)$ is not defined and $F^*(q', p)$ is defined. In our example, $\phi_{PO,br,BR}$ distinguishes PO from any other state of Z, $\phi_{PA,br,BR}$ distinguishes PA from any other state of Z, $\phi_{PC,br,BR}$ distinguishes PC from any other state of Z and $\phi_{CC,br,PO}$ distinguishes CC from any other state of Z, so $W = \{\phi_{PO,br,BR}, \phi_{PA,br,BR}, \phi_{PC,br,BR}, \phi_{CC,br,PO}\}$ is a characterization set of Z. Once a transition cover and a characterization set have been constructed, the test suite is given by the formula

$$S(\Phi^0 \cup \Phi^1 \cup \ldots \cup \Phi^{\beta+1})W,$$

where S is a transition cover, W is a characterization set, and (as already noted) β denotes the difference between the estimated maximum number of states of the implementation under test and the number of states of the model.

In order for the successful application of the test suite to guarantee the conformance of the implementation to the model, the SXM model has to satisfy two design for test conditions: output-distinguishability and input-completeness.

The set of processing functions Φ is called *output-distinguishable* if, for every two processing functions $\phi_1, \phi_2 \in \Phi$, if there exists $m, m_1, m_2 \in M$, $\sigma \in \Sigma$, $\gamma \in \Gamma$ such that $\phi_1(m, \sigma) = (\gamma, m_1)$ and $\phi_2(m, \sigma) = (\gamma, m_2)$ then $\phi_1 = \phi_2$. In our example, Φ is not output-distinguishable since, for example, both $\phi_{PO,br,BR}$ and $\phi_{PA,br,BR}$ produce the output BR while processing any memory value m and input br. The set Φ can be transformed into one that is output-distinguishable by suitably augmenting the output alphabet. In our running example we may enlarge Γ by considering as output for each transition a pair formed by both the current and the next state of the transition.

The set of processing functions Φ is called input-complete if, for every processing function $\phi \in \Phi$ and every memory $m \in M$, there exists an input symbol $\sigma \in \Sigma$ such that (m, σ) is in the domain of ϕ. In our running example, Φ is not input-complete since, for example, for $\phi_{CC,br,PA} \in \Phi$ and $po2cc \in M$, there is no input $\sigma \in \Sigma$ such that $(po2cc, \sigma)$ is in the domain of $\phi_{CC,br,PA}$. The set Φ can be transformed into one that is input-complete by suitably augmenting the input alphabet and the processing functions. In our running example, $\phi_{CC,br,PA}$ can be augmented by introducing an extra input symbol, say σ_e, and setting $\phi_{CC,br,PA}(po2cc, \sigma_e) = (\lambda, PA)$. Naturally, the extra inputs, outputs and transitions will be removed after testing has been completed.

6 Conclusions and Further Work

In this paper, we have presented our current work, focusing on an application of membrane computing to modelling and analysing engineering systems. As our initial attempt, we have considered the cruise control system of e-bike as our case study. We have modelled an e-bike cruise control system using kernel P systems and validated its behaviour using the kPWorkbench verification environment. We have also illustrated how the automata and X-machine testing methodologies can be applied on the kP model of the cruise control system.

As future work, we are planning to show how more complex engineering problems can be solved, tested and verified by using kP systems.

Acknowledgements

The work of SK is supported by Innovate UK (project no: KTP010551). MB is supported by a PhD studentship provided by the Turkey Ministry of Education. FI is supported by a grant of the Romanian National Authority for Scientific Research, CNCS-UEFISCDI, project number PN-III-P4-ID-PCE-2016-0210.

References

1. Bakir, M.E., Ipate, F., Konur, S., Mierla, L., Niculescu, I.: Extended simulation and verification platform for kernel P systems. In: Membrane Computing - 15th International Conference, CMC 2014. pp. 158–178 (2014)

2. Bakir, M.E., Konur, S., Gheorghe, M., Niculescu, I., Ipate, F.: High performance simulations of kernel P systems. In: 2014 IEEE International Conference on High Performance Computing and Communications, HPCC 2014. pp. 409–412 (2014)
3. Cimatti, A., Clarke, E.M., Giunchiglia, E., Giunchiglia, F., Pistore, M., Roveri, M., Sebastiani, R., Tacchella, A.: NuSMV 2: An opensource tool for symbolic model checking. In: Computer Aided Verification, 14th International Conference, CAV 2002, Proceedings. pp. 359–364 (2002)
4. Coakley, S., Gheorghe, M., Holcombe, M., Chin, S., Worth, D., Greenough, C.: Exploitation of high performance computing in the FLAME agent-based simulation framework. In: 14th IEEE International Conference on High Performance Computing and Communication, HPCC 2012. pp. 538–545 (2012)
5. Dragomir, C., Ipate, F., Konur, S., Lefticaru, R., Mierla, L.: Model checking kernel P systems. In: Membrane Computing - 14th International Conference, CMC 2013. pp. 151–172 (2013)
6. Gheorghe, M., Ceterchi, R., Ipate, F., Konur, S.: Kernel P systems modelling, testing and verification - sorting case study. In: Membrane Computing - 17th International Conference, CMC 2016. pp. 233–250 (2016)
7. Gheorghe, M., Ceterchi, R., Ipate, F., Konur, S., Lefticaru, R.: Kernel P systems: from modelling to verification and testing. Theoretical Computer Science (accepted for publication), `http://hdl.handle.net/10454/11720`
8. Gheorghe, M., Ipate, F.: On testing P systems. In: Membrane Computing - 9th International Workshop, WMC 2008. pp. 204–216 (2008)
9. Gheorghe, M., Ipate, F., Dragomir, C., Mierla, L., Valencia-Cabrera, L., García-Quismondo, M., Pérez-Jiménez, M.J.: Kernel P Systems - Version I. Eleventh Brainstorming Week on Membrane Computing (11BWMC) pp. 97–124 (2013)
10. Gheorghe, M., Konur, S., Ipate, F., Mierla, L., Bakir, M.E., Stannett, M.: An integrated model checking toolset for kernel P systems. In: Membrane Computing - 16th International Conference, CMC 2015. pp. 153–170 (2015)
11. Holcombe, M.: X-machines as a basis for dynamic system specification. Software Engineering Journal 3(2), 69–76 (1988)
12. Holcombe, M., Ipate, F.: Correct Systems: Building a Business Process Solution. Applied computing, Springer-Verlag (1998)
13. Holzmann, G.J.: The model checker SPIN. IEEE Transactions on Software Engineering 23(5), 275–295 (1997)
14. Ipate, F., Holcombe, M.: An integration testing method that is proved to find all faults. International Journal of Computer Mathematics 63(3-4), 159–178 (1997)
15. Konur, S.: An event-based fragment of first-order logic over intervals. Journal of Logic, Language and Information 20(1), 49–68 (2011)
16. Konur, S.: Specifying safety-critical systems with a decidable duration logic. Science of Computer Programming 80(Part B), 264 – 287 (2014)
17. Konur, S., Fisher, M.: A roadmap to pervasive systems verification. The Knowledge Engineering Review 30(3), 324341 (2015)
18. Konur, S., Gheorghe, M., Dragomir, C., Mierla, L., Ipate, F., Krasnogor, N.: Qualitative and quantitative analysis of systems and synthetic biology constructs using P systems. ACS Synthetic Biology 4(1), 83–92 (2015)
19. Konur, S., Kiran, M., Gheorghe, M., Burkitt, M., Ipate, F.: Agent-based high-performance simulation of biological systems on the GPU. In: 17th IEEE International Conference on High Performance Computing and Communications, HPCC 2015. pp. 84–89 (2015)

20. Lefticaru, R., Konur, S., Yildirim, U., Uddin, A., Campean, F., Gheorghe, M.: Towards an integrated approach to verification and model-based testing in system engineering. In: The International Workshop on Engineering Data- & Model-driven Applications (EDMA-2017) within the IEEE International Conference on Cyber, Physical and Social Computing (CPSCom). pp. 131–138 (2017), http://hdl.handle.net/10454/12322
21. Păun, G.: Computing with membranes. Journal of Computer and System Sciences 61(1), 108–143 (2000)
22. The P systems website. http://ppage.psystems.eu, [Online; accessed 30/10/17]
23. Sanassy, D., Fellermann, H., Krasnogor, N., Konur, S., Mierla, L., Gheorghe, M., Ladroue, C., Kalvala, S.: Modelling and stochastic simulation of synthetic biological boolean gates. In: 2014 IEEE International Conference on High Performance Computing and Communications, HPCC 2014, Paris, France, August 20-22, 2014. pp. 404–408 (2014)
24. Varadarajan, A.V., Romijn, M., Oosthoek, B., van de Mortel-Fronczak, J.M., Beijer, J.: Development and validation of functional model of a cruise control system. In: Proceedings of the 13th International Workshop on Formal Engineering Approaches to Software Components and Architectures. pp. 45–58. EPTCS (2016)