arXiv:1707.06858v1 [cs.SE] 21 Jul 2017

Mastering Heterogeneous Behavioural Models

J. Christian Attiogbé

LS2N - UMR CNRS 6004 - University of Nantes
Christian.Attiogbe@univ-nantes.fr

A detailed version of a paper accepted in the MEDI'2017 conference

Abstract. Heterogeneity is one important feature of complex systems, leading to the complexity of their construction and analysis. Moving the heterogeneity at model level helps in mastering the difficulty of composing heterogeneous models which constitute a large system. We propose a method made of an algebra and structure morphisms to deal with the interaction of behavioural models, provided that they are compatible. We prove that heterogeneous models can interact in a safe way, and therefore complex heterogeneous systems can be built and analysed incrementally. The Uppaal tool is targeted for experimentations.

Keywords: Behavioural models, Heterogeneous systems, Interaction

1 Introduction

Mastering the composition of heterogeneous models contributes to settle the challenge of building and analyzing large systems. The subsequent benefits are the gain of flexibility and an easier evolution of systems construction.

Models are used at different abstraction levels and for different purposes. Data models (also qualified as static models) capture the structure of manipulated data; behavioural models (also called dynamic models) often based on transition systems or event systems help to predict and reason about the behaviour of the software to be built. Other models such as timed models, security models, functional models are required according to the needs. A combination of these models is often necessary. In this article we focus on behavioural models often used to capture the evolution and the interaction between parts (called processes) of a more general heterogeneous system which can combine hardware and various software components.

An example of an heterogeneous system is an assembly of pieces of software and hardware communicating with a distributed architecture (using smart objects, sensors, actuators, mechanical parts driven by software). This kind of systems is spreading more and more as users adopt their provided services and practical facilities. However mastering their design, proving their correctness and maintaining these systems are a challenge of first importance for the security of services and software, and especially for reducing time to market of smart objects.

The integration of various formalisms inside a main one is often used as a solution to master the difficulties of heterogeneous systems. This constitutes an approach with a strong coupling of the formalisms; we dot not follow this trends, since we consider that a weak coupling favours reuse and flexibility.

There are several works and proposals related to heterogeneous issues; they embrace different abstraction levels and adopt various policies. Interface Theories are intensively studied [5]; they are based on reasoning on the contracts that link components and the information provided in their interfaces. SysML [7] adresses system engineering at modelling levels. SystemC [4] adopts a rather low abstraction level by composing software or hardware modules which are classes containing processes modelling functionalities. The core of SystemC consists of an event-driven simulator working as a scheduler. The Ptolemy project [9,6] proposes one of the most advanced framework, Ptolemy II [19] with which we share some concerns. But these are general purpose and heavy weight approaches which, from our point of view, constraint a lot the used components; they build a kind of a scheduler of the whole composition of components. We rather target a specific framework with a very weak coupling of components that can be composed (plugged) or unplugged at any time. For that, we consider the behaviours of the interacting components described with different formalisms, hence the heterogeneity. We especially focus on the communication channels that they use. We address systems with evolving adhoc structures, for small aperture net of components. With our approach the designer, before going further on development or during system maintenance, can for instance check if plugging a new component leads to a correct interaction or not.

This work is motivated by the necessity of light methods and tools to face the construction and the analysis of heterogeneous systems; these methods and tools should ensure flexibility, easy of use and evolution. The difficulties of heterogeneity arise not only at language level (data, property or behavioural), but also at the semantic level. The latter being more challenging.

We propose a method to make it easier the composition and hence the interaction between heterogeneous but compatible behavioural models. The idea is that one can easily compose models described with different formalisms but having the same compatibility domain, for instance their semantics are translated into transition systems. Currently we focus on such behavioural models. We implement a prototype tool, namely aZiZa, to support the experimentations with the method.

The article is organized as follows. Section 2 introduces the materials we have used. Section 3 is devoted to the proposed method, an algebra to structure the composition of models. Section 4 deals with a complete experimentation supported by the developed tools. Section 5 concludes the article.

2 Materials: Models Compatibility and Composition

Two main features have to be distinguished in terms of modelling: description formalisms and semantic models. First order logic, grammars, automata are

examples of description formalisms. Decision tables, labelled transition systems, operational semantic rules, axiomatic systems are examples of semantic models.

Heterogeneity is mastered at both levels; but we focus on semantic models which we consider as *compatibility domains*. Several categories of compatibility domains can be considered, for instance labelled transition systems, event-based models, predicate transformer *à la Dijkstra*.

We consider mainly the semantic models used by the modelling formalisms and extend them to the mathematical foundations they use. Automata and processes, as description formalisms, share the same semantic domain: Labelled Transition Systems (LTS).

Labelled Transition Systems Given a set of states S, a set of labels L, a labelled transition system is defined by the tuple $\langle S, L, \xrightarrow{l} \rangle$ where $\xrightarrow{l}$ is a transition relation: $\xrightarrow{l}: S \times L \to S$

Product of Transition Systems The parallel composition of processes, can be built by the *free product* of the transition systems of the processes. Theoretically, their free product results in a model where a global state[1] is made of a state of each of the processes, a global transition from a global state is made of the transitions of each of the processes from its state involved in the global state. However, practically, the global transitions can be constrained to avoid inconsistency in the access or update of the state variables.

In general free product is appropriate to the composition of independent (asynchronous) processes which may share communication channels. Besides, the *synchronous product* is appropriate to highly synchronous processes.

In our work, the considered global systems are asynchronous and distributed; their components can be concurrent and dynamic and the communication channels can be synchronous or asynchronous.

Compatible Models Two models M_1 and M_2 (or more) are said *compatible* or not, with regards to at least three compatibility levels: syntactic compatibility, semantic compatibility and formal-reasoning compatibility.

Syntactic compatibility can be solved at description formalisms level with (syntactic) model transformations known as *shallow embedding* [10]; semantic compatibility permits a *deep embedding* [10] of one model into the other provided that there are a compatibility domain: when the semantics of one model can be expressed in terms of the semantics model underlining the other model; the semantic compatibility is also used as the basis for formal reasoning.

From the formal point of view, first order logics and higher order logics provide logical frameworks with powerful constructs to describe and reason on heterogeneous objects, whatever their nature.
Transition Systems [2], Mealy Machines [18], with their various extensions, are

[1] The term *global state* refers to the state of the process composition

widely used to handle complex dynamic systems and are at the heart of many analysis methods and verification tools. The underlying theories are well-studied and, in the current state of the art a lot of effective systems are *compiled* into transition systems. Process Algebra (such as CCS [15], CSP[17,16], LOTOS[13], π-calculus [14]) built on top of transition systems are recognized as powerful behavioural description models; they are also representative of many behavioural languages, hence their use as composition and interoperability basis.

Definition 1. *(Compatibility Domain) A Compatibility Domain is defined as a category of models characteristics in such a way that, any two models considered within this domain, are comparable w.r.t. the considered characteristics. It is a model integration basis.*

Examples of *semantic compatibility domains* are: logics, labelled transition systems, trace semantics, temporal logics, weakest preconditions, Kripke model, algebra. For instance first order logic enables reasoning on several formalisms which use logic for objects description; when object descriptions (from different formalisms) have been translated as predicates in first order logic, the resulting predicates can be combined by the operators of first order logic. Examples of *syntactic compatibility domains* are grammars, set-theory, flowcharts.

Compatibility domains are the basic frameworks to compose or to integrate models within a global system.

Proposition 1. *Within a compatibility domain, it is always possible to translate objects semantics (from one formalism and paradigm) into the domain, to compose or integrate them, to reason within the basis, and to possibly translate results in target formalisms.*

The practical use in our work of compatibility domain is the definition of bridges between models and their semantics. We need the definition of *model bridges* between models in syntactical and semantic levels. Roughly, the principle of a bridge is to embed the model underlying a description into a given semantic model which will be used at model composition level.

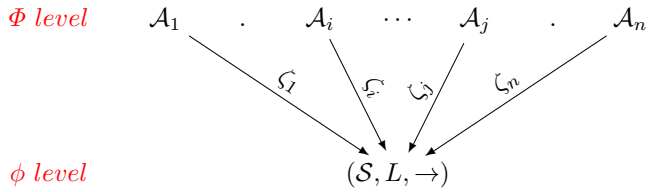
Semantic Models and Semantic Embedding A direct application of the notion of compatibility is the construction of semantic bridges between models or the semantic embedding of one model into another one. We choose the LTS as a reference behavioural model, because it is widely used and equipped with various tools.

If two models M_1 and M_2 are in a compatible semantic domain (LTS in our case), it exists structure morphisms ζ_1 and ζ_2 with related meaning matching such that $\zeta_1(M_1) = LTS_1$ and $\zeta_2(M_2) = LTS_2$.

Accordingly, we are about to define some operators Φ which arguments are different but compatible behavioural models; these operators form an algebra that leads the interaction of behavioural models. The idea is that $\Phi(M_i, M_j)$ is

semantically unfolded as $\phi(\zeta_i(M_j), \zeta_j(M_j))$ where ϕ is the domain-compatible equivalent of Φ.

If we consider a behavioural model M as a term of a given algebra $\mathcal{A}$, a sketch of the semantic embedding of models is as follows; we consider $\mathcal{A}_i$ as the source algebra to describe various but compatible models, then it exists a compatible domain denoted here by $(\mathcal{S}, L, \rightarrow)$ the LTS.



It follows that when models are compatible, a bridge can be used to relate them via the semantic structure induced by the compatibility domain (for instance their LTS). Consequently, a multilevel bridge can be gradually built between compatibility domains to link two or more models.

3 Interaction of Hererogeneous Models: an Algebra

Interaction between behavioural models, whatever their description formalisms, is viewed as exchanges through common communicating channels. Typically the interaction is denoted by a flow of emission and reception statements. Process algebra models, as a compatibility domain, capture very well these interactions, where the unit of specifications is a *process* expressing an elementary sequential behaviour; more complex behaviours are expressed with the composition (sequential, parallel, etc) of other processes, elementary or not.

Handling the heterogeneity is as simpler as if the LTS is the known user manual of each component. On the one hand, we extract the LTS from given components to compose them; on the other hand the LTS can be given by the component providers. Besides, an implementation can be built from a LTS used to tune a composition.

Therefore an abstraction of the communicating processes can be considered (see Fig.1) where the heterogeneity of the source models is tackled via the abstract behaviours captured with transition systems. According to Fig.1, if a, b and e are synchronous actions, then the interacting processes can for instance perform the trace: c.a*.b.e. We only need the communication actions of each process, not the entire behaviour. However, we must make precise the semantics of this interaction, and generalize the communication hypothesis; typically the relationship between synchronisation actions, and communication channels.
We define a set of operators that impact the behaviour of process composition:

- a process composition can be restructured through the renaming of channels;
- process communications can be broken through the modification of channels;
- the structure of a complete net of processes can evolve through channel restructuring, etc

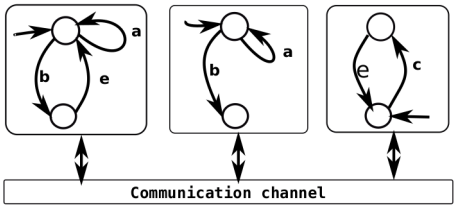


Fig. 1: Interaction between the processes: model level

3.1 The Core Operators for Model Interaction

We are about to elaborate an algebra $\mathcal{A}$ to structure and analyse the composition of heterogeneous processes. The operators of the algebra are related to the two levels (Φ level and ϕ level) considered in Section 2, while the structure set of the algebra is the set of processes $\mathcal{P}$ to be composed. Our target is an algebra $\mathcal{A} = \langle \mathcal{P}, O_\Phi, O_\phi \rangle$; therefore we introduce these sets of operators.

In the following, a *process* is denoted by the term:

Process procName [channel parameters](other parameters) {body}

where we consider its name, its channels and parameters, and a body. The body is an LTS which describes the behaviour of the process. Several named instances of a process can be defined using the process name as a type.

A *system* is made with the composition of at least two processes.

compose*: abstract parallel composition of processes* Let P_1 and P_2 be two processes which use a shared communication channel nc.

Process Proc[nc]() P1,
Process Proc[nc]() P2.

The expression $S = \mathsf{compose}(P_1, P_2)$ is the system made of the parallel composition of the processes P_1 and P_2 which interact via their common nc channel. Note that a channel can be hidden in a process by the renaming of the channel. The arity of the compose operator is not a strong constraint; a set of n processes can be composed either with the binary composition

$$\mathsf{compose}(\ldots(\mathsf{compose}(\mathsf{compose}(P_1, P_2), P_3), \cdots), P_n)$$

or directly with the list of processes as arguments: $\mathsf{compose}(P_1, P_2, \cdots, P_n)$.

The compose operator is an instance of the Φ operator. Typically, the embedding functions ζ_i compute the transition systems from the processes used as arguments of compose; then ϕ is the synchronous product [16] of the resulting transition systems.

A component process of a system built by the compose operator may be **selected** with the projection operator denoted by $\uparrow$. Consequently an operation α can be applied to a process inside a composition by selecting it as follows: $\alpha(\mathsf{compose}(P_1, P_2, \cdots, P_n) \uparrow P_3)$.

rename*: renaming a channel in a process* The expression (P rename c as nc) denotes a process P where the channel c is renamed as nc.

Let P_3 be a process using nc as a channel: Process Proc[nc]() P3. The expression $S = \mathsf{compose}(P_1, P_2 \text{ rename } nc \text{ as } c, P_3)$ results in a system where only P_1 and P_3 interact through nc. The behaviour of P_2 does not impact the behaviour of S since P_2 uses a local channel, thus the behaviour of P_2 is ignored in S.

replace*: substitution of processes* Within a system, a given process is substituted by a given new one. The replace operator needs three arguments: a system S, a process oP already in S, a new process nP not in S. The process oP should share its channels with S. The process nP should have the same shared channels (for the substitution) but it can have more channels.

The effect of the replacement is based on the shared channels; the shared channel oP is cut and replaced by the common channel in nP. The expression $sys = \mathsf{replace}(Sys, oP, nP)$ modifies Sys by replacing inside it, the behaviour of oP by the new behaviour expressed by nP.

Formally the channels shared by Sys and oP are renamed in oP with a new name unused in Sys and nP. Then Sys is composed with nP. Consequently if nc is the channel shared by the three processes, c a fresh channel, then we have:

$\mathsf{replace}(Sys, oldP, newP) = \mathsf{compose}(Sys, (Sys \uparrow oP) \text{ rename nc as c, nP})$

remove*: removing a process from a composition* A given process can be removed from a system. The remove operator (symbolically denoted by $\downarrow$) requires two arguments: a system S made at least with two processes, a process P already part of S. The process P will be removed from S; this is symbolically denoted by $S \downarrow P$. For instance the expression $sys = \mathsf{remove}(\mathsf{compose}(P_1, P_2, P_3), P_2)$ results in a system composed of the processes P_1 and P_3.

extractChan*: listing the channels of a process* This operator, when applied to a process, gives the list of channels used inside the process. The channels of processes can then be compared, reused, renamed, hidden.

These operators constitute our target algebra and practically a core language: $\langle \mathcal{P}, \{\mathsf{compose}, \mathsf{replace}, \mathsf{select}, \mathsf{remove}\}, \{\mathsf{rename}, \mathsf{extractChan}\}\rangle$. It is expressive enough, to describe the composition and the interaction between behavioural models as illustrated in the next section.

3.2 Illustration: a Heterogeneous Control System

We consider the interaction between a net of processes modelling a control system equipped with sensors, actuators and controlers. These devices come with their different behaviours, from various vendors. From our heterogeneous modelling point of view the interfaces of the models are defined textually as follows and can be viewed as depicted in Fig. 2. Note that the precise behaviours of the devices are not required at this stage. The communication channels at the interfaces are enough to define the interaction.

```
Process Sensor[channel ic]() {...}          S1,
Process Controler[channel ic, cc]() {...}   C1,
Process Actuator[channel cc]() {...}        A1.
```

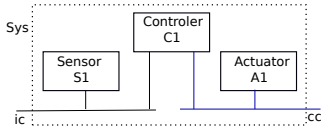


Fig. 2: Interaction net of processes

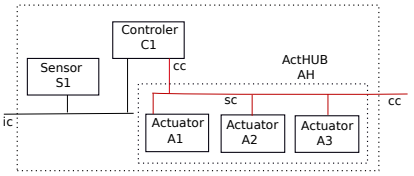


Fig. 3: Reshaped net of processes

Let a process controller C_1 interacting by reading a channel ic and writing on a control channel cc. Let a process Sensor S_1 interacting by sending data on the same channel ic. At the modelling level, we could simply write compose(C_1, S_1) so that C_1 and S_1 interact via ic. Considering A_1 as the actuator process interacting by reading the channel cc, then the description $sys =$ compose(compose$(C_1, S_1), A_1)$ builds a new system (see Fig. 2) where the three processes interact together through the channels ic and cc. The controler C_1 may send orders to the actuator A_1 depending on data read from S_1. Now, we define two actuator processes A_2 and A_3 and a hub of actuators HA which sends its data to A_1, A_2 and A_3:

```
Actuator A2
Actuator A3
HA = compose((A1 rename  cc  as  sc), A2, A3)
```

The behaviour expressed by replace$(sys,\ A_1,\ (HA$ rename sc as $cc))$ results in a new system (see Fig. 3) where the controler C_1 is not anymore directly connected to A_1 but to HA via the channel cc, a renaming of the previous sc channel of HA. In the same way we can easily add new sensors S_2, S_3 into an existing pool of sensors with the compose operator: compose(sys_1, S_2, S_3). The three sensors will write on the channel ic.

3.3 Semantics of Interaction

The common interaction level considered here is related to the communications between the composed processes. As introduced in Section 2, a compatibility domain is required to ensure consistency. Moreover, to reach this compatibility domain, from heterogeneous models, some filters could be applied. Typically compatible models have various additional specific facets. For instance dealing with the domain of transition systems, some models may be structured according

to multiple layers related to: communication channels, time properties, time constraints, QoS, $\cdots$.

Consequently, the models should be filtered according to these specific facets (time, channels, properties, etc) in order to get the desired compatibility and interaction level. At this stage we consider only the communication facet. Therefore the interaction between composed models are based on their communication through the used channels. The other facets will be consider latter in our work.

As far as the interaction between the behavioural models is considered, each process evolves according to the channels it uses. Interaction is based on communication via shared channels using emission and reception mechanisms (message passing). Synchronous channels involve handshake communications. A reception takes place when a process applies the appropriate reception primitive relatively to a channel and, there is a (abstract) data sent on the addressed channel by the emission primitive applied by another process. In the case of asynchronous channel, if there is nothing on the channel, the attempt of reception is aborted.

3.4 Extending the Core Operators

The previously defined core operators can be extended to compute other heterogeneous processes and systems. But more specifically, analysis operators should be defined to manipulate the built systems.

Consider that we have a system made of sensors, actuators, controlers and many other smart devices, making an adhoc network of communicating processes. We would like to plug a new device in the system so that it can interact with the existing processes; for instance a new plugged sensor detects the existing controlers and sends data to them, or a new plugged actuator joins the system and becomes ready to interact with the existing processes which send orders to the actuators. But, it is not reasonably feasible to directly experiment with the system prior to the consistency decision. Therefore reasoning at model level is appropriate. The system builder may evaluate the forthcoming system, decide if some components or operations are correct or not before performing them on the real system. This is profitable if the used models and operations on models are trustworthy. Consequently we would like to easily check the consistency of the new composition of processes prior to implementing it. For instance, with the aim of getting a diagnosis (at least True or False) of the envisaged composition, we would like to write the following:

$$\mathsf{check}(compose(sys, newSensor))$$
$$\mathsf{check}(replace(sys, oldProcess, newProcess))$$

These scenario motivate the need to define the check operator which is not a process composition operator but an analysis one. Typically this kind of operators should implement at least the interaction compatibility, the absence of deadlock, liveness property. In the current stage of the work we reuse for this purpose, the existing tools of process algebra: UPPAAL [3] which has its own

graphical input description formalism, SPIN [11] which has the Promela language as input process description language and CADP [8] which uses Lotos as input process description language.

4 Practical Analysis of Heterogeneous Models

We report on a case study dedicated to a distributed control system. The system consists of a set of robots which supervise a geographically widespread area and take actions with respect to events in the area: intruders or found unusual objects. Sub-components of the system are responsible of patrolling in different parts of the area and looking for preassigned objects; in case of detection of such objects a signal is sent to a supervisor. Other sub-components follow a specific object or a detected intruder and communicate its location to the supervisor.

This kind of system is representative of systems being designed or used to assist people in hospitals, homes, remote care, restaurants, smart future factories, etc. Indeed when constructing or maintaining large distributed systems, it is easy to collect components everywhere and combined them to build the systems. But when reliability should be tackled, we have to consider at least the models of the various components. These models are merely built with the appropriate description models instead of a common one for all the components. Hence an heterogeneity of the models at concrete or abstract level.

As far as our case study is concerned, at the design level, if we focus on the behavioural aspect related to the interaction between the system and its components, the system can be modelled as the combination of several processes; each one having specific features and a specific behaviour.

The target global system can be built incrementally with various components provided that the requirements are known. It is what we have experimented with. We consider the following components, which were built by considering the requirements of the case study.

A component, routine **patroller**, modelled by a process (RPatrol) which, after a connection to a supervisor, sends every δ seconds its collected data to the supervisor. This component was modelled using Uppaal because we need time feature provided by Uppaal.

A **scrutineer** component, modelled by a process (Scrutineer) which, after a connection to a supervisor, gets from the supervisor its roadmap made of target objects to look at, and then moves around and when one of the target object is found, it sends a signal to the supervisor. We modelled this component using Promela which provides high expressivity for handling data and control.

A **data collector** component (dataCollector) which, after a connection to a supervisor, becomes ready and sends on demand its collected data to the supervisor. This component is modelled as a simple transition system with the DOT formalism, seen as a common language used by several transition system formalisms. We give an excerpt in Fig. 4 for illustration purpose. Note the use of the synchronisation actions connection and readState, which will be renamed latter using the related operator of our kernel. As depicted in Fig. 5 (the process

named `dtcltr1`) we can see the process as systematically renamed and translated into Uppaal by the tools we have built.

For experimentation purpose, we built several modules to implement the core operators from the DOT format: channel renaming, process selection, process removing and specific facet filtering (time, ...).

A **follower** component is dedicated to detect and follow intruder or assigned object. This component first try a connection to the supervisor; on success it scruts around and then follows the assigned object. The relative positions are then sent to the supervisor until the followed object leaves or when the supervisor orders to stop. We have specified this component as a LOTOS process.

```
digraph p_dataCollectorBot {
 initSCRT -> S2 [label= connection!];
 S2 -> initSCRT [label=KO?];
 S2 -> S3 [label=OK?];
 S3 -> S4 [label=ready!];
 S4 -> initSCRT [label=stop?];
 S4 -> S5 [label= readState!];
 S5 -> S4 [label=getState?];
}
```

Fig. 4: A LTS model of the data collector in the DOT format

A supervisor or **global controler**, modelled as a process is written in Uppaal.

4.1 Combining the Heterogeneous Processes

We use the proposed operators to experiment with the composition of the previous processes. For instance Fig. 5 depicts the composition resulting from the term: compose($dtctrl1$ rename connection as connect, spv, $rpt1$) where:

- $dtctrl1$ is an instance of the process dataCollector after two renamings and a translation into Uppaal:
  ((dataCollector rename connection as connect) rename readState as sendState)
- $rpt1$ is an instance of the patroller RPatrol,
- spv is an instance of supervisor.

Several ζ_i morphisms were used to embed the provided models into LTS. The DOT formalism have been intensively used.

- $\zeta_{dot2upa}$ A translator from DOT to Uppaal using XML as the internal representation of the Uppaal tool.
- $\zeta_{pml2upa}$ A translator from promela to Uppaal. We reuse the features of SPIN; from Promela we generate a DOT format of a process using the features provided by SPIN. Then we reuse our module that embed DOT into Uppaal.

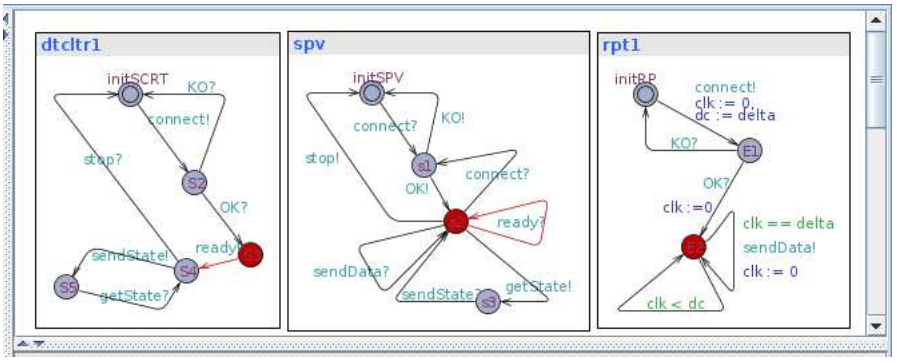


Fig. 5: Interaction between composed processes of the system

– $\zeta_{dot2lotos}$ A transator from DOT to Lotos process.

The proposed algebra and method for composing heterogeneous processes enable us to deal with several examples. In case of examples where the communications actions do not bear data the basic structure morphisms are easily implemented and work well. In case where we have a complex structuring of data used as parameters of communication actions, the structure morphisms are not straightforward, since specific interpretations can be given to the data; an action such as chan!value may denote not simply the sending of `value` on the channel but the sent value may also conveying a semantic such as a process identifier.

The various experiments also reveal that a more expressive formalism is preferable to be used as the target of embedding in order to avoid loosing information. For this purpose we target the Lotos language for further experiments.

4.2 Analysis of the Global System

Remind, one interest of gathering the various models is being able to perform an analysis of the global system. In the current case of behavioural models, as stated before, at least standard liveness properties should be analysed on the basis of the global system. We have been able to perform such analysis on the basis adopted for the composition.

Deadlock. The absence or presence of a deadlock is a standard property. It is expressed in LTL logic provided by the Uppaal tool as follows:

```
A[] not deadlock
```

Progress. Specific progress or reachability properties can be stated using the Uppaal model. For instance we check that both the data collector and the routine patroller interact simultaneously with the controler and can reach some given states of their behaviour. This is expressed as follows:

```
E<> dtctrl1.ready  and  rpt1.E2
```

Examples of other properties we have checked are depicted in Fig. 6

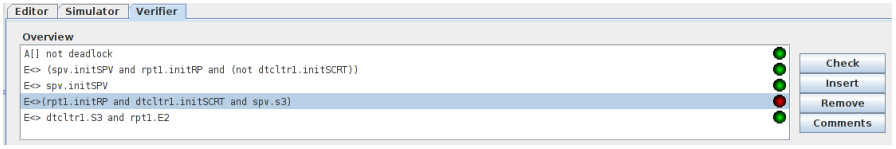


Fig. 6: Examples of checked properties

4.3 Tool Support: aZiZa

To experiment with examples, as shown in the previous section, we have developed the main modules of a prototype tool called aZiZa[2], to support our method proposal. This is necessary to validate the proposed concepts and to improve the global composition method. The core modules are described bellow.

Dot to Uppaal	A translator between an LTS formatted with DOT and the XML repremsentation of Uppaal processes
Promela to Uppaal	A translator from Promela to Uppaal. Here we reuse the dot generator of the Spin tool. Then we combine with the previous translator.
Label Filtering	This module is a filter to extract a part of a dot LTS
Channel renaming	This module is used to rename a channel of a process
Promela process splitting	It splits a complete promela system into several processes. This module is developed for the purpose to reuse a given process.
Process selection	This module achieves the selction of a target process from a given composition.

Yet, all the steps are not automated. Some steps are still manual, for instance the result of the translation of the process, is to be copy-paste in a Uppaal file, or the graphical layout are not automated.

The architecture of the prototype tool is depicted in Fig. 7. Its is made of interconnected modules (the blocks in the figure). As for now we intensively use intermediate files to connect the blocks in oder to make easy the independent development of the blocks.

[2] `aziza.ls2n.fr`

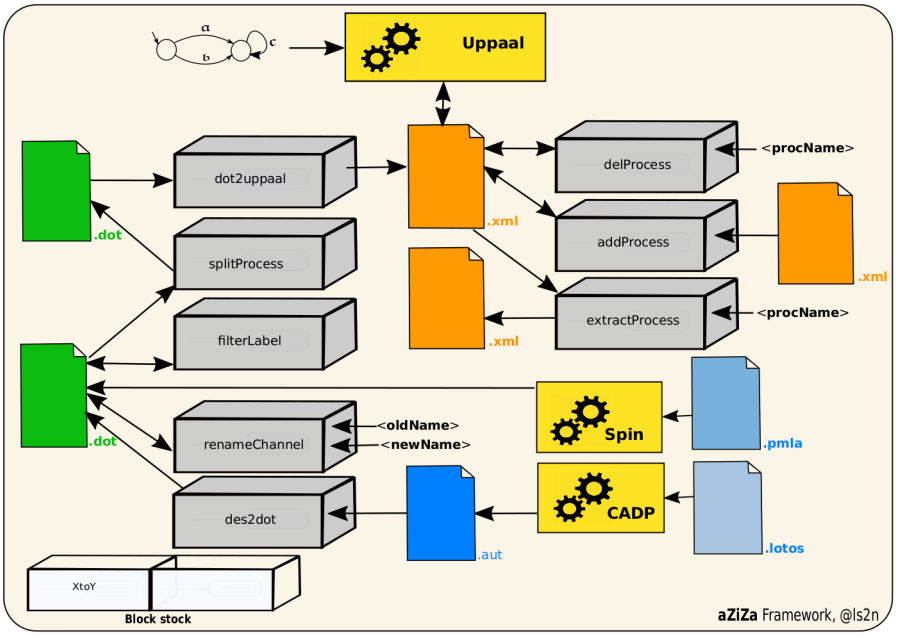


Fig. 7: The architecture of the aZiZa tool

5 Conclusion

We have shown that under the hypothesis of semantic domain compatibility, the composition of heterogeneous behavioural models can be overcomed. We have used labelled transition systems as common semantics domain to found our method of composition. The method is based on an algebra of operators that focuses on the manipulation of channels which are the communication mechanism between the composed models. We have equipped the method with tools in order to experiment with case studies. We illustrated the article with a case study of a distributed control system where various processes cooperate to control objects evolving within a given area. The case study reported here is a part of a series of experiments that serve as a mean of assessment of the proposal and a testbed to improve the tools under development.

One representative of the related works is Ptolemy II [12,19]. Ptolemy achieves the interaction between different actor-oriented models using an abstract semantics (namely the actor semantics). It also enables the use of finite state machines in place of actor-oriented models, but the interaction works rather as a global scheduler, controling a sequential execution flow of the FSM considered each as a global state linked via a port to another one. Moreover Ptolemy II is a general purpose heavy weight composition framework. Unlikely we target a specific, flexible and extensible framework dedicated to the composition and analysis of behavioural models dedicated to the growing small aperture nets of processes.

There are some aspects that are not yet considered at this stage of the work; for example we have considered some components dealing locally with time constraints but dealing with time constraints at the global level is a challenge. As future work, we have planned experimentations with the more expressive CADP framework and especially its exp.open composition tool. We plan some improvements among which the propagation of global properties inside local components and vice versa. For this purpose we are investigating the Property Specification Language [1], an IEEE standard, as a pivotal for property passing through the components and through the various tools.

References

1. IEC 62531 Ed. 1 (2007-11) (IEEE Std 1850-2005): Standard for Property Specification Language (PSL). *IEC 62531:2007 (E)*, pages 1–152, Dec 2007.
2. André Arnold. Verification and Comparison of Transition Systems. In M-C. Gaudel and J-P. Jouannaud, editors, *TAPSOFT'93: Theory and Practice of Software Development, CAAP/FASE*, volume 668 of *LNCS*, pages 121–135. Springer, 1993.
3. Gerd Behrmann, Alexandre David, and Kim G. Larsen. A Tutorial on Uppaal. In M. Bernardo and F. Corradini, editors, *4th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM-RT 2004*, number 3185 in LNCS, pages 200–236. Springer–Verlag, 2004.
4. D.C. Black, J. Donovan, and and A. Keist B. Bunton, editors. *SystemC: From the Ground Up, Second Edition*. Springer, 2010.
5. Luca De Alfaro and Thomas Henzinger. Interface Theories for Component-based Design. In *Embedded Software*, pages 148–165. Springer, 2001.
6. Johan Eker, Jörn W. Janneck, Edward A. Lee, Jie Liu, Xiaojun Liu, J. Ludvig, Stephen Neuendorffer, S. Sachs, and Yuhong Xiong. Taming Heterogeneity - the Ptolemy Approach. *Proceedings of the IEEE*, 91(1):127–144, 2003.
7. Sanford Friedenthal, Alan Moore, and Rick Steiner. A Practical Guide to SysML. The MK/OMG Press. Morgan Kaufmann, Boston, 2015.
8. Hubert Garavel, Frédéric Lang, Radu Mateescu, and Wendelin Serwe. CADP 2011: a Toolbox for the Construction and Analysis of Distributed Processes. *STTT*, 15(2):89–107, 2013.
9. Antoon Goderis, Christopher X. Brooks, Ilkay Altintas, Edward A. Lee, and Carole A. Goble. Heterogeneous Composition of Models of Computation. *Future Generation Comp. Syst.*, 25(5):552–560, 2009.
10. M.J.C. Gordon. *Introduction to HOL: A Theorem Proving Environment*. Cambridge University Press, 1993.
11. G. J. Holzmann. The Spin Model Checker. *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997.
12. Edward A. Lee. Disciplined Heterogeneous Modeling. In D. C. Petriu, N. Rouquette, and Ø. Haugen, editors, *13th International Conference, MODELS 2010, Oslo, 2010*, volume 6395 of *LNCS*, pages 273–287. Springer, 2010.
13. LOTOS. *A Formal Description Technique Based on The Temporal Ordering of Observational Behaviour*. IOS - OSI, Geneva, 1988. International Standard 8807.
14. R. Milner, J. Parrow, and D. Walker. A Calculus of Mobile Processes. *Journal of Information and Computation*, 100, 1992.
15. Robin Milner. *Communication and Concurrency*. Prentice-Hall, 1989.

16. A. W. Roscoe and J. Davies. CSP (Communicating Sequential Processes). In David A. Padua, editor, *Encycl. of Parallel Computing*. Springer, 2011.
17. A.W. Roscoe. *The Theory and Practice of concurrency*. Prentice-Hall, 1998.
18. C. H. Roth and L. L. Kinney. *Fundamentals of Logic Design*. Thomson, 2004.
19. S. Tripakis, C. Stergiou, C. Shaver, and E. A. Lee. A Modular Formal Semantics for Ptolemy. *Mathematical Structures in Computer Science*, 23(4):834–881, 2013.