MODELLING VARIABLE-STRUCTURE SYSTEMS
PH.D. PROPOSAL

ERNESTO POSSE

Abstract. A variable-structure system is a system whose structure changes over time. This includes systems whose topology evolves (such as mobile systems) and systems which change their *mode* of operation as the result of some event (such as hybrid systems.) Such systems are generally more difficult to understand and analyze than systems with a static structure. Therefore we need a solid basis for modelling, simulating and reasoning about them. In this thesis we focus on timed, reactive systems and we propose an approach to modelling these systems based on the Statechart and DEVS formalisms. We investigate the capabilities of these formalisms to model variable-structure systems, and the relationships between them.

1. Introduction

Scientific and Engineering disciplines rely on understanding (and designing, in the case of Engineering) complex systems. This implies a need for modelling and analyzing, and in some cases simulating, these systems. Complexity arises in many different ways such as the size of the system, the apparent or real lack of regularity of its behaviour, or the diversity of components. A basic aspect which also contributes to the complexity is variable structure.

Variable structure systems are systems whose structure changes over time. Typical examples of systems with a changing structure are mobile communications networks, mobile (code) agents, chemical compounds, dynamic geographic information systems, and many systems which exhibit some form of adaptability to environmental change.

Since modelling, simulation and analysis are concerned with system behaviour, the central question is that of the *semantics* of the underlying formalisms used to describe the dynamics of systems. Any proposed framework, whether theoretical or practical, must be founded on a precise, sound and preferably complete semantics of the intended domain.

The objective of this thesis is to develop a well-founded framework for the modelling, simulation and analysis of variable-structure systems.

In this framework, systems are described by models in the Statechart formalism [10, 11], the DEVS formalism [36], and a new formalism which combines the features of these. We study if and how variable structure systems can be modelled by these formalisms. We also study the relationships between them, specifically from a formalism-transformation point of view in order to support a unifed semantic view as well as simulation environments. In addition to these, we try to establish the meaning of "variable structure" in the formalisms of interest and whether this notion is preserved by formalism-transformations.

A central issue that we address is that of compositionality of these formalisms and their semantics, in order to maintain a modular approach to systems modelling and design.

The rest of this document is organized as follows: section 2 discusses some general aspects concerning Modelling, Analysis and Simulation. Section 3 briefly presents existing languages and formalisms used to model variable-structure systems. Section 4 focuses on the Statechart and DEVS formalisms, as well as a language we have developed called Devslang. Section 5 presents a sketch of the new proposed approach. Finally, section 6 summarizes the thesis objectives, the contributions made so far, and the proposed contributions.

2. Modelling, Analysis and Simulation

A framework for modelling, simulation and analysis has the purpose of understanding an existing system, or designing a new one. In both cases we start with the construction of a model of the existing reality or the system to be engineered.

2.1. **Modelling.** The process of *modelling* is the process of creating a *model*, this is, a description of some reality, *system* or *object of interest*, usually with the purpose of understanding it, and extracting useful information from it. The model must contain enough information to capture those aspects of the system in which we are interested. As a description, it needs to be given in some language or formalism. This language must be expressive enough to represent the features of interest of the object or system modelled, and to give enough relevant information about the system.

There are many different formalisms with distinct features, and therefore one must make a careful decision about which one is the appropriate for the system at hand. In spite of such diversity of formalisms, there seems to be a certain class of formalism capable of capturing the behaviour of a wide range of systems. These are the so-called *discrete-event* formalisms. We focus in this thesis on such formalisms. A discrete-event system is one where the set of possible events is discrete, the behaviour is time-dependent, and the state is piece-wise constant (i.e., between any two events, whether internal or external, the state does not change.) Such formalisms can describe many common *discrete-time* systems (events occur only at "clock ticks".) Continuous-time systems are often dealt with using discretization techniques from Numerical Analysis. A useful formalism for discrete-event systems is DEVS (see [36].)

Any language or formalism has a syntax and a semantics. In some cases, these may not be fully defined, but the more precisely defined these are, the more information we can get out of our models. Hence we favour the use of well-defined formalisms in a modelling, analysis and simulation framework.

The syntax and semantics of the formalism may refer to both structural and behavioural aspects of a system. In some cases, the semantics may be purely structural, as is the case, for example, with UML class diagrams. In others, the semantics defines the behaviour of the system, as is the case with FSA's or Petri Nets.

2.2. **Analysis.** The process of *analysis* has as purpose the understanding of a system of interest and more concretely, establishing properties of the system. Reasoning about a system is often contingent on the modelling formalisms used, as well as the reasoning formalisms used (i.e., the *logic*.) Hence the selection of formalisms is also critical in this part of the process.

We distinguish between two kinds of properties: system-specific properties (e.g., this system will deadlock,) and the more general formalism-wide properties (e.g., any model with certain features will have a periodic behaviour.) The first kind of property would preferrably be determined by automatic means such as a model checker. The second approach requires either manual proof or automatic theorem provers capable of dealing with universal statements over models.

There is a diverse set of techniques to obtain such proofs. A major technique is *well-founded induction*, and in particular induction on the structure of the system or the derivation of certain statement. This technique relies however, on inductively defined sets for the structure of the system and/or its state. If the models are described in a non-inductive fashion, as is the case with graphical formalisms, the techniques can be difficult to apply or not applicable at all. In such cases other techniques are necessary.

When addressing system specific properties through model checking, the selection of a logic language is crucial. There are many different logics apt to reason about the behaviour of dynamic systems. Temporal logics such as LTL, CTL, CTL* (see [14]) and the μ-calculus [15] are particularly useful. These logics however, deal only with an abstraction of time in terms of computational steps. It is desirable to have a logic that allows reasoning with respect to *external time*, this is, a notion of time separate and independent of the system.

Analyzing a system depends on the semantics of the formalism used to model it (and the formalisms used to reason about it.) There are different approaches to semantics. The most important approaches are *denotational* and *operational* semantics. In the first, the meaning of a model is given by associating

it with an element in some abstract domain. The properties of such domain can then be used to reason about the systems modelled. This approach to semantics is sometimes too abstract, and difficult to use for instance, in the case of concurrent systems. Operational semantics describes the meaning of a system in terms of concrete behaviour (e.g., how it would be simulated by some abstract machine,) which usually makes it more intuitive and easy to use. It also serves as a basis for the construction of interpreters and simulators.

Denotational semantics usually focuses on an abstract view of the system (e.g., what is the function computes by the system,) while operational semantics is more concrete (e.g., how behaviours are generated.) Since the ideal is to obtain a complete understanding of the system, analysis should be done at different levels of abstraction. Therefore both approaches to semantics are significant and useful. But their usefulness depends on the semantics being consistent with each other. We say that the semantics of a formalism or language is *fully-abstract* if its denotational semantics and operational semantics agree on the properties of interest.

2.3. **Simulation.** There are many systems whose features make them difficult to analyze by symbolic means (e.g., complex non-linear systems with chaotic behaviour, or systems with an extremely large state space.) In such cases we have to resort to simulation in order to obtain an understanding of the system's behaviour. The process of *simulation* has as its purpose to generate instances of a system's behaviour.

Simulation algorithms are formalism-specific, as they depend on the operational semantics of the formalism. Nonetheless, a wide range of simulation algorithms seems to be closely related to discrete-event algorithms. Furthermore, the discrete-event approach can be related to common techniques for operational semantics, as shown in section 4. By focusing on such algorithms we obtain not only great deal of generality in a simulation framework, but also the possibility of applying techniques to reason about the operational semantics of the simulated systems.

Discrete-event simulators often rely on event queues, and therefore can be related to Dataflow Networks as well. There are different approaches, from purely sequential simulation to parallel and distributed algorithms. However, the concurrency of the model is independent of the concurrency of the simulator. It is often convenient to think of the components of a model as a set of concurrent or parallel processes, even if the simulator is sequential. In such cases, a time-sharing mechanism is used by the simulator.

Modelling, analysis and simulation based on solid principles and techniques provide a rich framework to understand and develop complex systems. In the rest of this section we discuss some general but important topics that can serve as the basis for those principles and techniques.

2.4. **Multi-paradigm modelling.** Some systems are complex enough so that their description using a unique formalism is too difficult in practice. This is usually the case of the so-called *heterogeneous systems*, that is, systems made up of multiple components where each component is of a substantially different nature than the others. To model such systems, a preferred approach is *multi-formalism modelling*. In this approach, we use a different formalism for each component, together with some sort of "glueing" formalism to assemble the components.

In multi-formalism modelling, the different formalisms may be used to represent not only structural components, but also different "aspects" or "views" of a system. The typical example is the UML, which consists of several formalisms, such as class diagrams to represent system structure, Statecharts, Petri Nets, and Interaction diagrams to represent system behaviour.

In all cases, the main issue is defining a unified semantics for a system made up of components or views in disparate formalisms. One approach to deal with this is formalism translation. We can give the semantics of such systems, but translating each component to a common formalism, as long as it preserves the features of interest. Here, proving that such translations preserve the properties of interest is fundamental, and, as mentioned above, there are numerous techniques, including induction. As before, induction may not be applicable if the translation is not in the form of a recursive function. This can be the case if we use *graph-grammars* [8] to translate graphical models.

Another way of tackling complexity is to look at a system at *different levels of abstraction*, in order to extract only whatever information is relevant to us and that help us better understand the system under scrutiny.

Apart from using multiple formalisms, and multiple levels of abstraction, we can also resort to modelling the formalisms themselves. This is known as *meta-modelling*, and provides us with a mechanism to deal uniformly with multiple formalisms. These three techniques are collectively called *multi-paradigm modelling*. [24]

2.5. **Hierarchy of System Specification.** The specification of a system by a model can be done at different levels of abstraction. In [36], Zeigler defines a hierarchy of system specification for timed, reactive systems, and includes both structural and behavioural aspects. This hierarchy makes a few assumptions. Since it is for reactive systems, systems are assumed to have input and output ports through which the system interacts with its environment. Furthermore, since the systems are dynamical, the concept of time is also associated to the system's behaviour.

2.5.1. *Time bases and signals.* The notion of time is formalized as a *time-base*, that is, a tuple $(T, <)$ where T is a partially ordered set (and usually a total linear order) w.r.t. $<$. Furthermore, the time-base is usually equipped with a commutative addition operation over T, and an element ∞ such that for all $t \in T$ $t + \infty = \infty$. Sometimes it may also have a minimum element t_0. It is usually required for addition to be monotonic w.r.t. $<$, that is, if $t_1 < t_2$ then $t_1 + t < t_2 + t$ for all $t \in T$. Given a time base T, we define the open and closed time intervals $T_{(t} \stackrel{def}{=} \{t' \mid t < t'\}$, $T_{t)} \stackrel{def}{=} \{t' \mid t' < t\}$, $T_{[t} \stackrel{def}{=} \{t' \mid t \leq t'\}$, $T_{t]} \stackrel{def}{=} \{t' \mid t' \leq t\}$, $T_{(t_1,t_2)} \stackrel{def}{=} \{t' \mid t_1 < t' < t_2\}$, etc.

Given some set A, a *time function*, also known as *trajectory* or *signal*, over A is a partial function from a time base T to A. The restriction of a time function to an interval is called a *segment*. We denote $\Omega(T, A)$ the set of all segments over A with time base T. We assume that such set is equipped with a concatentation operation $\bullet$ between contiguous segments, where two segments f_1 and f_2 are *contiguous* if the least upper bound of the domain of f_1 is the same as the greatest lower bound of the domain of f_2. It is often desirable for a set $\Omega(T, A)$ to be closed under concatenation.

2.5.2. *Levels of abstraction.* The hierarchy can be summarized as follows, ordered from the highest level of abstraction to the lowest:

(1) I/O observation frame
(2) I/O relation
(3) I/O function
(4) State/Transition
(5) Network

The first level can be said to be purely definitional. The next three are concerned with behaviour, and the last is purely structural.

At the observation frame, only the time-base T, the input set X and output set Y are specified. Hence a system is given by the triple (T, X, Y). To represent individual ports we can give the sets X and Y more structure, and make them, for instance, labelled product sets. We call the pair (X, Y) the *interface* of the system.

At the I/O relation level, a system is specified by a triple $(T, X, Y, \Omega_{in}, R)$ where (T, X, Y) is an observation frame, $\Omega_{in} \subseteq \Omega(T, X)$ and $R \subseteq \Omega_{in} \times \Omega(T, Y)$ such that if $(f, g) \in R$ then $dom(f) = dom(g)$.

At the I/O function level, a system is specified by a triple $(T, X, Y, \Omega_{in}, F)$ where $(T, X, Y, \Omega_{in}, F)$ is an I/O relation frame such that F is a function. It is worth noticing that at this level, the system is considered deterministic: given an initial state and an input segment, there is a unique output segment.

At the state/transition level, the system specifies the internal states and how the state changes with respect to input segments. In this case a system is given by a tuple $(T, X, Y, \Omega_{in}, Q, \Delta, \Lambda)$ where (T, X, Y) is an observation frame, $\Omega_{in} \subseteq \Omega(T, X)$, Q is a set of states, $\Delta : Q \times \Omega \to Q$ is a global state transition function, and $\Lambda : Q \times X \to Y$ is an output function, such that Ω_{in} is closed under left segmentation (for every $f \in \Omega_{in}$, if $t \in dom(f)$ then $f_{t>} \in \Omega_{in}$ where $f_{t>}$ is the restriction of f to the

subset of $dom(f)$ less or equal to t,) and satisfies the "composition property" that for all contiguous input segments f_1 and f_2 and all states q, $\Delta(q, f_1 \bullet f_2) = \Delta(\Delta(q, f_1), f_2)$. We note that this definition does not consider non-deterministic systems. We could consider an alternative hierarchy, where Δ is a relation rather than a function.

The final level is that of network, where the system is given by a tuple (T, X, Y, G) where (T, X, Y) is an observation frame and $G = (M, C, s, t, Z)$ is a graph where M is a set of state transition systems (the subcomponents,) C is a set of connections, $s, t : C \to M$ are the source and target functions defining the graph's topology, and $Z : C \to \{Z_l | Z_l : Y_{s(l)} \to X_{t(l)} \text{ for each } l \in C\}$ is a set of transfer functions where X_i and Y_i are the input and output sets of component $i \in M$.

The hierarchy outlined above is of course, only one of several possible. In particular, we could have a system specified at the level of state/transitions and still be non-deterministic.

2.5.3. *Systems morphisms.* A fundamental aspect of modelling and analysis is to establish relations between different system specifications. There are many such possible relations. One particularly important kind of relation establishes whether a system captures the "essence" of another in the sense that either one is embedded in the other, or one is represented by the other. This kind of relation is particularly important from the point of view of modularity and verification. Whenever we try to determine if a given component can replace another, we need such a relation, or *morphism* between the two systems.

Associated with each level of the hierarchy we have several possible notions of morphisms between systems.

At the I/O observation frame level we can define both structural and behavioural morphisms. A structural morphism at this level which captures the ability of a system to replace another could be defined in terms of having a *compatible* interface. There are several possible definitions of compatibility. A basic one is simply to have the same input and output sets. More generaly we could require a map between the corresponding sets (including the time-base.) If we consider these sets as *types*, this could be generalized even further by a notion of *subtyping*. If X_2 is a subtype of X_1 and Y_1 is a subtype of Y_2 then any system with interface (X_1, Y_1) can replace a system with interface (X_2, Y_2).[1] This defines a preorder relation necessary for the ability of A to replace B. We could go further if we considered these sets structured to represent ports. A behavioural morphism at this level between two specifications (T, X, Y) and (T', X', Y') should relate the possible sets of input and output signals, for example, such a morphism could be given by a pair of mappings $g : \Omega(T, X) \to \Omega(T', X')$ and $k : \Omega(T, Y) \to \Omega(T', Y')$ where k is surjective. Other definitions are possible.

At the I/O relation and function levels, we require a mapping between the corresponding relations. Perhaps the most basic is containment: there is a morphism from A to B if $IOR(A) \subseteq IOR(B)$, where $IOR(X)$ denotes the I/O relation of X. This could be generalized to any mapping from $IOR(A)$ to $IOR(B)$. Mappings at this level need not preserve all properties. For instance, a map could scale the rate of output signal of one of the systems with respect to the other.

At the level of state/transitions, the notion of replaceability can be naturally captured by the notion of similarity (and equivalence by bisimilarity, see [20, 21, 26].) A can replace B if A simulates B. The morphisms at this level are usually simulations or bisimulations, but there are several other morphisms of interest, such as graph-homomorphisms, monotonic and continuous mappings (with respect to some preorder of states,) and contraction mappings (with respect to some metric between states.)

At the network level, a useful morphism is that of graph-homomorphism. Establishing that such homomorphism exists between two networks means that the target network can be seen as an abstraction of the source in the sense that several components are lumped together in one component. At this level, the notion of compatibility would require specific structural contraints (e.g. A can replace B if A contains a subcomponent C that satisfies a particular constraint.)

Some particular morphisms for this hierarchy are discussed in more detail in [36].

The system specifications at a level together with the corresponding morphisms can be described as *categories* [5]. To our knowledge, the categorical view of this particular hierarchy has not been explored.

[1] This is analogous to the standard rule for subtyping of functions in the typed λ-calculus.

2.5.4. *Abstraction and realization.* Different levels of system specification can be related by abstraction morphisms (going up) or refinement or realization relations (going down.) When designing a system the modeller is likely to start at the higher levels, and add information along the way, leading to further refinements. It is key then to ensure that a system specified at a lower level indeed satisfies the specification at the higher level.

To ensure that a system at a lower level satisfies a specification at a higher level it is not enough to provide a mapping. The mapping should also preserve the morphisms of interest at the lower level, otherwise, the abstraction does not really capture the concrete systems. For example, a mapping from the state/transition level to the input/output relation level should be such that if A simulates B (where A and B are state/transition specifications,) then $IOR(B) \subseteq IOR(A)$. If we consider each level of abstraction a category, an abstraction mapping should be a *functor* [5] (possibly contravariant) between the corresponding categories.

We expect the same from a realization mapping: it should not simply map abstract specifications to concrete implementations but it should preserve the morphisms present at the higher level.

What should be the relation between abstraction and realization? If abstraction and realization are functors, we could be tempted to consider them the "inverse" of each other, but this is too restrictive. If we map a system to an abstraction, this could have many different realizations. We think that a more appropriate relation between an abstraction and a realization should be that of an *adjunction* [5].

2.5.5. *System equivalence and compositionality.* A particularly important kind of system morphism is that of *equivalence*. System equivalence plays an essential role in modularity, abstraction and verification. By establishing the equivalence of two systems at some level of abstraction, we can not only replace one by the other in any context, but also, obtain abstractions at the same level of specification, and reason about a large class of systems. Indeed, if we partition a family of systems in equivalence classes, then all properties preserved by the equivalence will be satisfied by all systems in a given equivalence class, and therefore whatever property of interest we establish about a given system, will hold for all its equivalent systems. Such equivalence classes can be considered an abstraction of its elements. For this reason, any reasonable notion of equivalence has to be defined in such a way that it preserves any properties of interest at the corresponding level of abstraction.

Preserving properties of interest is not a sufficient condition for an equivalence relation to be considered a truly appropriate notion of system equivalence. Systems are usually part of larger systems. Two systems should be considered equivalent if they are indistinguishable from the point of view of their environment. If two systems A and B are equivalent, then any context $C[-]$, should not distinguish between A and B, and therefore $C[A]$ must be equivalent to $C[B]$ (where $C[X]$ represents putting the system X in place of the placeholder $-$ in the context $C[-]$.) In other words, the equivalence relation must be a *congruence*. We call such equivalence relations *compositional*.

As any morphism, the definition of system equivalence depends on the level of abstraction in the hierarchy. There are a few obvious choices. At the input/output relation level, the natural notion is that of equality, or bijection between the input/output relations. At the state/transition level, bisimilarity is the usual choice. Nevertheless, the simplest choice is not always adequate. Consider for example Kahn's process networks [19]. If we define the equivalence of these networks, as the equivalence between their input/output relations but abstracting the time of the events, and the systems are non-deterministic, then the equivalence is not compositional as a result of the so-called Brock-Ackermann anomaly (see [25].)

Compositionality also depends on what we consider to be contexts. If we are dealing with a structural equivalence relation, the context of interest would be purely structural (determined by the syntax of the model.) But if we are talking about behavioural equivalence such as bisimilarity, the context needs to take into account the system's state. In some formalisms, the two can be identified (see for instance the π-calculus in section 3,) but in general this is not the case.

In general, any well-founded framework for modelling, analysis and simulation must have well defined morphisms at the appropriate levels of abstraction, in particular it must have reasonable notions of

equivalence. Such relation should be compositional. Furthermore, there should be suitable abstraction and realization mappings which preserve equivalence. These are the features we look for in this thesis.

3. Existing approaches to modelling variable-structure

Describing a system as a variable-structure system depends on what we mean by "structure". One definition of "structure" is "the aggregate of elements of an entity in their relationships to each other," or "an arrangement in a definite pattern of organization." Under this definition, the notion of system structure is fundamentally that of system composition: how a system is composed of subsystems and how these are related to each other. This is essentially a topological view of structure. With this view a system with variable structure is a system that changes in its pattern of composition and/or its pattern of connections between components.

An alternative view is to call "structure" whatever describes a system's behaviour. In this case, a variable-structure system is a system where such description changes, and therefore the system's behaviour also changes.

In this section we look at these two forms of variable structure systems and some of the existing approaches to model them.

3.1. Topological change.

When we consider structure to be determined by the composition of a system and the relationships or connections between components, we have a fundamentally graphical view of the system. In such a view, there are different ways in which the structure can change, such as:

- Creation and Destruction:
  - Creation of new components
  - Creation of new links (relations or connections between components)
  - Destruction of components
  - Destruction of links
- Mobility:
  - Change of links: Moving of a link from one component to another
  - Change of nesting:
    - Moving a component inside another component
    - Moving a component outside its current "parent" component

Many of these operations can be defined in terms of others. For instance, change of nesting can be defined in terms of change of links, by observing that nesting can be seen as a special kind of link. Moving can be described in turn in terms of destructing and creating components and/or links.

All these operations have been dealt with to some extent in different languages and formalisms. We identify the following as the fundamental approaches which have been proposed to represent topological change of structure:

- Object Oriented Programming
- The π-calculus
- The Ambient Calculus
- Graph-grammars
- Bigraphical Reactive Systems
- DS-DEVS/DS-System Networks

Common to all these is a graph-based notion of structure, where this notion could be plain graphs, hypergraphs, higraphs, or any other variant of graphical structures.

3.1.1. *Object Oriented Programming.*

The paradigm of Object Oriented Programming (OOP) [1] can be seen as providing a high-level abstraction for representing variable structure, where the system being modelled is represented by an object together with its aggregates, i.e. each component is modelled itself by some object. The aggregation relation defines the structure of the system. In this view, OOP describes change of structure by means of creation and destruction of objects.

The underlying graphical view of OOP is straightforward and supported by modern notations such as object diagrams in the Unified Modelling Language (UML [33, 9].) Objects are the nodes of the

graph, and aggregation (and similar relations) defines the edges. In languages supporting nested classes, a more accurate descritpion could be given by some variant of Higraphs.

3.1.2. *The π-calculus.* A fundamental approach was proposed by Milner, Parrow and Walker (see [23]) in the form of a process algebra intended for mobile concurrent processes. Mobility in the π-calculus, addresses the change in the connections between different components, that is, the topology of the network of communication channels. Here systems or processes are modelled by terms. Processes are connected by channels which have a name. A process has two fundamental operations: sending a message along a channel and receiving a message along a channel. Mobility arises by allowing messages to be channel names, thus enabling processes to communicate their channels to other processes which then gain access to them.

While processes are represented as terms, they can be described graphically, where each node represents a process in a particular state, and each edge represents a channel. More precisely the structure here is that of a hypergraph, since a channel can be shared by more than two processes.

The semantics of the π-calculus is given in the style of Structural Operational Semantics (SOS [27]), where the meaning of a process is given by a *state transition system*. This gives rise to a rich theory developed around the notion of *behavioural equivalence*, a crucial concept from the point of view of compositionality, and verification.

There are many variants of the π-calculus, considering different forms of communication. Of interest to us, is the so-called Higher-Order π-calculus [34], where processes themselves can be communicated as messages. This model could be more closely represented by a form of Higraphs, and makes it closer to the Ambient calculus, discussed next.

3.1.3. *The Ambient calculus.* Closely related to the π-calculus is Cardelli and Gordon's Ambient calculus (see [7].) This is also a process algebra for mobile concurrent systems, but mobility in this case addresses the change in nesting relations rather than channel connections. Systems are also modelled by terms. Each process or *ambient* has a name identifying it, and an *area* or *domain*, where subprocesses are located. The three primitive operations are: getting inside some process, getting outside, and "openning" a domain (i.e. destroying a domain's boundaries.)

In the pure form of the calculus, the language does not have channels explicitly. Nonetheless, it is provably equivalent in expressive power to the π-calculus, as channel-based communication can be easily simulated by moving pure ambients.

The graphical description of Ambients can be given by Higraphs without edges, and labels for each node.

Its semantics is also given in the form of SOS, associating a state transition system to each process. There are also appropriate notions of behavioural equivalence.

3.1.4. *Graph-grammars.* When we view of the structure of a system as a graph (hypergraph, or higraph), we realise that variable structure can be described by specifying changes in subgraphs of a given graph. This is precisely captured by the notion of graph-grammar (see [8]) which extends the traditional notion of term grammars from Formal Language Theory to elements with richer structure than strings.

Informally, a graph-grammar is a collection of *rules* or *productions*, with a left-hand side (LHS) and right-hand-side (RHS) which are both graphs (hypergraphs, higraphs, resp.) A graph-grammar is applied to a *host-graph*, by executing one (or more) rules. A rule is applied if its left-hand-side matches a subgraph of the host-graph. In that case the subgraph is replaced by the right-hand-side. A computation is a sequence of rule applications.

The precise meaning of "matching" and "replacing" give rise to different variants and approaches. The two traditional algebraic approaches are called *Double Pushout Approach* (DPO) and *Single Pushout Approach* (SPO,) both based on the category-theoretical notion of *pushout*.

Further variants have been proposed by giving graphs a richer structure. This is the case of attributed graphs, and typed-graphs, both closely related to the notion of *meta-modelling*.

Graph-grammars can be seen as a model of variable structure where the change in structure is explicitely described by the rules. Graph-grammars can also be used for other purposes, in particular, .they can be used as a mechanism to describe transformations between models and formalisms.

3.1.5. *Bigraphical Reactive Systems.* Milner has proposed an alternative form of graphs-grammars called Bigraphical Reactive Systems (BRS) (see [22]), where systems are also represented by a *Bigraph* (which can be seen as a variant of a Higraph), and system dynamics are described by rules in the same style as graph-grammars. There are nonetheless some fundamental differences. Bigraphs allow the specification of "reactive" and "passive" contexts, which can control which rules are applicable and where can they be applied. The other fundamental difference is that the theory of Bigraphs has focused on the notion of behavioural equivalence, and in particular on how to obtain a labelled transition system from a BRS such that bisimulation is a congruence, i.e. an appropriate notion of behavioural equivalence. These ideas have not been fully developed by the standard graph-grammar community. On the other hand, the theory of standard graph-grammars has focused on studying the properties of graph-grammars and combinations of rules.

Several basic formalisms for transformational, interactive and reactive computation such as the λ-calculus, Petri Nets, the π-calculus and the Ambient calculus have been faithfully modelled by BRS.

3.1.6. *DS-DEVS/DS-System Networks/Dynamic I/O-Automata.* A different approach is characterized by the DS-DEVS formalism (see [6]), or more general by what we call DS-System Networks, closely related to Dynamic I/O-Automata (see [4]). A System Network is a collection of components connected through channels, similar to the π-calculus or to Kahn's process networks. Each component in turn can be made of subcomponents. The leaves in the nesting tree are called *atomic* components. The behaviour of an atomic component can be described by some formalism, usually a variant of a state-transition system. The meaning of a network is given by an equivalent atomic component.

To model variable structure, a system in the DS-System Networks formalism is specified by two levels: an *executive component*, and a set of System Networks. The executive is itself a system network, possibly atomic. The set of networks represents the set of all possible structures which the overall system can be in. Each state of the executive is associated with a specific network in the set. In this way, we could say that the executive controls the structure of the entire system.

In these formalisms, the change of structure is explicitly modelled by the executive. A main difference between this approach, and that of graph-grammars, is that the behaviour of executive, being itself a system, could depend on external stimuli, whereas a graph-grammar is a closed entity: the behaviour of a system is the computation that the graph-grammar generates, but this computation depends exclusively on the host-graph. Only if we encode stimuli as part of the host-graph or if we consider variants of graph-grammars do we obtain a similar notion of external interaction, which is fundamental to build systems compositionally.

3.2. **Mode change.** A different view of structure is that of model-structure, this is, structure is the form, shape, or syntax of the model which describes the behaviour of a system.

In this view, a system can be described by some state-transition system, where the structure of interest is the structure of this transition system. A variable-structure system can be said to be in different *modes* of operation. When the system is in a particular mode, the behaviour of the system is determined by a specific transition system. Change of mode is therefore change of structure. The part of the system that describes the change of mode is analogous to the "executive" as described in 3.1.6 and it can depend on external stimuli.

Two classical examples of such variable structure systems are Hybrid automata and Statecharts. We defer Statecharts to section 4.

3.2.1. *Hybrid Automata.* A hybrid system combines continuous-time behaviour with discrete-event or discrete-time behaviour. At any moment in time, the behaviour of the system can be described by a set of differential equations, but specific events may cause the system to change its mode of operation to a new, different, set of differential equations.

We can model such systems by means of a state-transition system where each state defines a mode and thus has a set of differential equations associated. This formalism is commonly known as *Hybrid Automata* (see [32].)

4. Statecharts, DEVS and Devslang

In this section we describe the formalisms which we use as a basis to our proposed formalism for variable-structure systems. We pay particular attention to DEVS and study in detail its operational semantics and its compositionality properties.

4.1. **Statecharts.** The Statechart formalism proposed by Harel [10, 11], can be viewed as an extension of finite state automata that supports nesting of states. Statecharts use Higraphs to represent nesting. These states are sometimes called *meta-states*, or simply *blobs*. Hyperedges denote transitions, and they may cross nesting boundaries. In addition, it supports *orthogonal components*, which provide a form of concurrency. A state contains zero or more orthogonal components, each of which has a Statechart. One can think of orthogonal components as concurrent processes within a state, which communicate by broadcasting. The meaning of a Statechart is given by an equivalent "flat" state atomaton, i.e. a state-transition system with no nesting, hyperedges or orthogonal components.

Statecharts naturally model a form of mode-based variable structure. A mode would simply be a meta-state containing some sub-Statechart. Transitions between these meta-states would represent change of mode. An important aspect is that since there are no restrictions on the source and target of hyperedges, these transitions can occur from any level of nesting to any other level. This allows to model mode-changes which depend not only on events but on substates as well.

There are many variants of this formalism, which differ on the semantics. The main two variants are STATEMATE Statecharts [13] and Rhapsody Statecharts [12]. It is worth noticing that the Rhapsody Statecharts are considered the "executable core" of the UML, and therefore are closely related to Object-Oriented Programming. More concretely, a Rhapsody system consists (at runtime) of a set of objects whose behaviour is described by a Statechart, where methods are represented by state transitions. Communication occurs by sending an event to an object. The event is placed in a queue for the object, and when processed, the event is broadcast within the object's Statechart. Each object can therefore be considered a concurrent process, in a manner similar to Actors (see [2, 3].)

The semantics of Statecharts have been given in the form of an algorithm. This is unfortunate since it is difficult to use some common techniques to reason about them. There have been some formalizations of these semantics (see [28]) but unfortunately these turn out to be non-compositional. Certain authors have obtained compositionality but at the cost of restricting some features of the full formalism (see [17, 18, 35]).

4.2. **DEVS.** DEVS [36] is not a formalism intended to model variable structure system, but to model timed, reactive, discrete event systems with a piecewise-constant state space. It is a formalism that enjoys considerable popularity in the Modelling and Simulation community, due to many of its attractive features, mainly, its support for modular development, its potential for parallelization and the existence of efficient simulation algorithms. Here we describe the so-called *Classic DEVS* variant in detail and present our compositionality result.

4.2.1. *Definition of DEVS.* In the DEVS formalism, systems or models are described as a collection of one or more *components*. There are two types of components: *atomic* (or *behavioural*) components and *coupled* (or *structural*) components. An atomic component defines a simple system that has a state, accepts input, produces output, and whose behaviour depends on external stimuli, the state, and the time the system spends on a state. A coupled component is basically a network of components (atomic or coupled,) which communicate through unidirectional asynchronous channels (possibly with multicasting.) A component may have ports, which play the role of channel connectors. We first introduce these notions without explicit reference to ports.

Definition 4.1. An *atomic DEVS component* A is a tuple $(X, Y, S, s_0, \delta^{ext}, \delta^{int}, \tau, \lambda)$ where X is a set of *input* values, Y is a set of *output* values, S is a set of *states*, $s_0 \in S$ is the *initial state*, $\delta^{int} : S \to S$ is the *internal transition function*, $\tau : S \to \mathbb{R}_0^+$ is the *time advance*, $\lambda : S \to Y \cup \{\bot\}$ is the *output function*,

and $\delta^{ext} : Q \times X \to S$ is the *external transition function*, where $Q \stackrel{def}{=} \{(s,e) : s \in S \wedge 0 \leq e \leq \tau(s)\}$ is the *total state space*.

Given such an atomic component A, define $inset(A) \stackrel{def}{=} X$, $outset(A) \stackrel{def}{=} Y$, $states(A) \stackrel{def}{=} S$ and $initial(A) \stackrel{def}{=} s_0$.

Informally, an atomic DEVS works as follows: at any moment in time, the system is in some state $s \in S$. The system remains in this state for an interval of time $\tau(s)$, if no external input is received. At this time, the system will produce output $\lambda(s)$ (an output of $\perp$ represents no output.) Then the system will jump to the state $\delta^{int}(s)$, and continue in the same way. However if external input is received at some time t_{inp} between the time t_s when the system entered state s and $t_s + \tau(s)$ then the system will jump to state $\delta^{ext}((s,e),x)$ where $e = t_{inp} - t_s$ is the time elapsed since the last transition, and $x \in X$ is the value of the input. In this case, no output is produced. Output is produced only when an internal transition takes place. If there is a conflict (i.e. $t_{inp} = t_s + \tau(s)$) then the external transition takes precedence over the internal transition.

Note that there are no terminal states, and that the system is *reactive*, in the sense that whenever there is input the system will perform a transtion, even if it means ignoring its input. Also note that there future state is fully determined by the current state, the time spent in the state, and the input if any.

Definition 4.2. A *coupled DEVS component* B is a tuple $(X, Y, N, C, infl, Z, sel)$ where X is a set of *input* values, Y is a set of *output* values, N is a set of *subcomponent names* including a special name "self", C is a set of *subcomponents* (atomic or coupled) indexed by N such that $B \notin C$, $infl : N \to 2^N$ is the *influencer function*, $sel : 2^N \to N$ is the *select function*, and Z is a set of *transfer functions*, namely

$$\begin{aligned} Z \quad &\subseteq \quad \{Z_{i,j} : Y_i \to X_j | i,j \in N \text{ and } i \in infl(j)\} \\ &\cup \quad \{Z_{\mathsf{self},k} : X \to X_k | \mathsf{self} \in infl(k)\} \\ &\cup \quad \{Z_{k,\mathsf{self}} : Y_k \to Y | k \in infl(\mathsf{self})\} \end{aligned}$$

where for each $i \in N$, X_i and Y_i are respectively the input and output sets of subcomponent $C_i \in C$.

Given such a coupled component B, define $inset(B) \stackrel{def}{=} X$, $outset(B) \stackrel{def}{=} Y$, $names(B) \stackrel{def}{=} N$, and $parts(B) \stackrel{def}{=} C$.

In the previous definition, a coupled component is seen as a network of components C, connected through "channels", specified by the influencer function $infl$ and the family of transfer functions Z. For a component named n, $infl(n)$ is the set of components whose output is an input of n. In this case, there is a function $Z_{i,n} : Y_i \to X_n$ for each influencer $i \in infl(n)$ which specifies how the outputs of i are to be transformed into inputs of n. The overall coupled component may be an influencer to some of its components. This is done by having the label $\mathsf{self} \in N$, and the transfer functions $Z_{\mathsf{self},k}$. This represents the fact that input to the overall component is transmitted to some subcomponents. Similarly, some subcomponents may be influencers of the overall component. The corresponding transfer function is given by $Z_{k,\mathsf{self}}$. Notice that a given component may be influencer of more than one component.

Informally a coupled component works as follows. We think of the component as the parallel composition of the subcomponents. This is, the subcomponents run concurrently and independently. When a subcomponent generates output, this is communicated asynchronously to all its influencees (applying the appropriate transfer functions.) This includes the overall component: if it receives external input, it is transmitted to the subcomponents k for which $\mathsf{self} \in infl(k)$. Similarly, if $k \in infl(\mathsf{self})$ for some subcomponent k, and k generates output, then the overall coupled component generates output. The subcomponents however are not trully concurrent in the sense that if at some time t there are two or more subcomponents which are supposed to perform a transition, only one of them executes it. The subcomponent which executes the transition is chosen by the selection function sel. If $imm \subseteq N$ is the subset of conflicting components, then the component chosen is $sel(imm)$. The set imm is called the *imminent* set.

There are a few other restrictions on DEVS components. First, we will consider only DEVS components whose time-advance is not zero, and coupled components $(X, Y, N, C, infl, Z, sel)$ such that there is no $m \in N$ for which $m \in infl(m)$, this is, no self loops are allowed.

Every DEVS system corresponds to a system in the hierarchy outlined in section 2.5. See [36] for details.

In what follows, let $ADEVS$ denote the set of all atomic DEVS components, $CDEVS$ the set of all coupled components and $DEVS \stackrel{def}{=} ADEVS \cup CDEVS$. We use $\mathbb{R}_0^+ = \{x \in \mathbb{R} | x \geq 0\}$ and $\mathbb{R}_{0,\infty}^+ = \mathbb{R}_0^+ \cup \{\infty\}$.

4.2.2. *Operational Semantics.* The formal meaning of DEVS components can be given in terms of the hierarchy of system specifications described in section 2.5 (see [36] for details,) but such semantics are far detached from an operational view of a system's behaviour. One of our main contributions at this point is an operational semantics for DEVS in the form of a *Labelled Transition System* (or LTS for short.) A labelled transition system over a set S of states and a set A of labels is given by a tuple $(S, A, \rightarrow)$ where $\rightarrow \subseteq S \times A \times S$ is a transition relation. We write $s \xrightarrow{a} s'$ for $(s, a, s') \in \rightarrow$ to mean that there is a transition from state s to state s' by a. By providing an inductively defined operational semantics in the form of an LTS, we are able to apply well-known techniques to reason about DEVS, and have a formal description which is more closely related to the way DEVS systems are supposed to execute.

Our definition of the operational semantics consists of two parts: *single transitions* and *executions*. In single transition semantics, the meaning of a DEVS component is given by an LTS where the labels are individual events (internal or external), and transitions represent what *could* happen (as opposed to representing what *would* happen.) The execution semantics defines how a DEVS component will behave given an input segment, i.e. a sequence of input events. The execution semantics is defined in terms of the single transitions.

The labels of the LTS representing a DEVS are of two forms: $\mathsf{ext}(t, x)$ for external events occurring at time $t \in \mathbb{R}_0$ providing an input $x \in X$, and $\mathsf{int}(t, y)$ for internal events occurring at time $t \in \mathbb{R}_0$ generating an output $y \in Y$. We denote $Evts(A)$ the set of possible events of component A.

The nodes of the LTS are *configurations*. A configuration of a DEVS component is a pair of the form (s, t) where s is the state of the component and t the time of the last transition. We denote $Configs(D)$ for the set of all possible configurations of D. For a coupled DEVS component D with $C = \{C_n : n \in N\}$ its set of subcomponents with names N, the state is a mapping $\rho_D : N \rightarrow SubConfigs(D)$, where $SubConfigs(D) \stackrel{def}{=} \cup_{K \in C} Configs(K)$, and $\rho_C(n) \in Configs(C_n)$. The state of a coupled component can be seen as a tree, with the same structure as that of the component.

Definition 4.3. The operational meaning of an atomic component $A = (X, Y, S, s_0, \delta^{ext}, \delta^{int}, \tau, \lambda)$ is given by an LTS $(Configs(A), Evts(A), \rightarrow_A)$ where $\rightarrow_A \subseteq Configs(A) \times Evts(A) \times Configs(A)$ satisfies the following for all A-configurations (s, t_l):

(i) Internal transitions (AIT):
$(s, t_l) \xrightarrow{\mathsf{int}(t, \lambda(s))}_A (\delta^{int}(s), t)$ if $t = t_l + \tau(s)$

(ii) External transitions (AET):
$(s, t_l) \xrightarrow{\mathsf{ext}(t,x)}_A (\delta^{ext}((s, t - t_l), x), t)$ if $t \leq t_l + \tau(s)$

To define the meaning of coupled components we need to take care of choosing the component that has precedence.

Definition 4.4. The *imminent set* of a coupled component B at a coupled state ρ is the set of components whose next internal event is scheduled sooner than any other component:

$$imm_B(\rho) \stackrel{def}{=} \{n \in N : \forall \rho' \in Configs(B).\rho(n) \xrightarrow{\mathsf{int}(t,y)}_n \rho'(n) \Rightarrow (\forall \rho'' \in Configs(B).\forall m \in N.\rho(m) \xrightarrow{\mathsf{int}(t',y')}_m \rho''(m) \Rightarrow t \leq t')\}$$

With this definition we can now define the small transitions for a coupled component.

Definition 4.5. Let $B = (X, Y, N, C, infl, Z, sel)$ be a coupled component. The operational semantics of B is given by an LTS $(Configs(B), Evts(B), \rightarrow_B)$ where $\rightarrow_B \subseteq Configs(B) \times Evts(B) \times Configs(B)$ satisfies the following for all B-configurations (ρ, t_l), where $\rho(n) = (s_n, t_n)$, $\rho' = (s'_n, t'_n)$ and $i^* \stackrel{def}{=} sel(imm_B(\rho))$:

(1) Internal transition (CIT): $(\rho, t_l) \xrightarrow{\mathsf{int}(t,y)}_B (\rho', t)$ if
   (a) $\rho(i^*) \xrightarrow{\mathsf{int}(t,y^*)}_{i^*} \rho'(i^*)$,
   (b) for each $n \in N$ such that $i^* \in infl(n)$ and $n \neq \mathsf{self}$, $\rho(n) \xrightarrow{\mathsf{ext}(t,x_n)}_n \rho'(n)$ where $x_n = Z_{i^*,n}(y^*)$,
   (c) for all $n \in N$ such that $n \neq i^*$ and $i^* \notin infl(n)$, $\rho(n) = \rho'(n)$,
   (d) and $y = Z_{i^*,\mathsf{self}}(y^*)$ if $i^* \in infl(\mathsf{self})$ or $y = \perp$ if $i^* \notin infl(\mathsf{self})$

(2) External transition (CET): $(\rho, t_l) \xrightarrow{\mathsf{ext}(t,x)}_B (\rho', t)$ if
   (a) for each $n \in N$ such that $\mathsf{self} \in infl(n)$ and $x_n \neq \perp$, $\rho(n) \xrightarrow{\mathsf{ext}(t,x_n)}_n \rho'(n)$, where $x_n \stackrel{def}{=} Z_{\mathsf{self},n}(x)$,
   (b) and for all $n \in N$ such that $\mathsf{self} \notin infl(n)$ or $x_n = \perp$, $\rho(n) = \rho'(n)$, where $x_n \stackrel{def}{=} Z_{\mathsf{self},n}(x)$

The execution semantics describes the behaviour of a component over time in terms of single transitions. In particular it describes the behaviour with respect to an input-segment, i.e. a sequence of external events.

Definition 4.6. *A partial execution* of a DEVS component M is a (possibly infinite) sequence

$$\overrightarrow{\gamma} = \langle \gamma_1, \gamma_2, \dots \rangle$$

where for each $i \geq 1$, $\gamma_i = ((s_i, t_i), \alpha_i, (s'_i, t'_i))$ such that each $\alpha_i \in Evts(M)$ and $(s_i, t_i), (s'_i, t'_i) \in Configs(M)$ satisfying:

(i) $(s_i, t_i) \xrightarrow{\alpha_i}_M (s'_i, t'_i)$
(ii) for all $i \geq 1$, if $\gamma_i = ((s_i, t_i), \alpha_i, (s'_i, t'_i))$ and $\gamma_{i+1} = ((s_{i+1}, t_{i+1}), \alpha_{i+1}, (s'_{i+1}, t'_{i+1}))$ then $(s'_i, t'_i) = (s_{i+1}, t_{i+1})$
(iii) for all $i, j \geq 1$, if $i \leq j$ then $time(\alpha_i) \leq time(\alpha_j)$,
(iv) and there is no $k \geq 1$ such that $time(\alpha_k) = time(\alpha_{k+1})$ and $type(\alpha_k) = \mathsf{int}$ and $type(\alpha_{k+1}) = \mathsf{ext}$.

We often write a partial execution $\overrightarrow{\gamma}$ as

$$\overrightarrow{\gamma} = (s_1, t_1) \xrightarrow{\alpha_1}_D (s_2, t_2) \xrightarrow{\alpha_2}_D \dots \xrightarrow{\alpha_{k-1}}_D (s_k, t_k) \xrightarrow{\alpha_k}_D \dots$$

Denote $start(\overrightarrow{\gamma}) \stackrel{def}{=} (s_1, t_1)$ the starting configuration, and

$$partials(M, (s,t)) \stackrel{def}{=} \{\overrightarrow{\gamma} | \overrightarrow{\gamma} \text{ is a partial execution of } M \text{ such that } start(\overrightarrow{\gamma}) = (s,t)\}$$

We call a DEVS state s a *sink* if $\tau(s) = \infty$ and $\delta^{ext}((s,e),x) = s$ for all $e \leq \tau(s)$ and $x \in X$.

A *total execution* is a partial execution $\overrightarrow{\gamma}$ which is either infinite or finite and whose last configuration is (s_n, t_n) such that s_n is a sink.

$$executions(M, (s,t)) \stackrel{def}{=} \{\overrightarrow{\gamma} | \overrightarrow{\gamma} \text{ is a total execution of } M \text{ such that } start(\overrightarrow{\gamma}) = (s,t)\}$$

The *event trace* $\overrightarrow{\alpha}$ of an execution is it's sequence of events $\alpha_1 \alpha_2 \dots \alpha_k \dots$.

$$trace(\overrightarrow{\gamma}) \stackrel{def}{=} \left\langle \alpha_i | (s_i, t_i) \xrightarrow{\alpha_i}_M (s'_i, t'_i) \in \overrightarrow{\gamma} \right\rangle$$

and

$$traces(M, (s,t)) \stackrel{def}{=} \{\overrightarrow{\alpha} | \overrightarrow{\alpha} \text{ is an event trace of some } \overrightarrow{\gamma} \in partials(M, (s,t))\}$$

A *timed-ordered event sequence* $\overrightarrow{\alpha}$ is sequence of events $\langle \alpha_1, \alpha_2, \ldots \rangle$ such that for all $i, j \geq 1$, $i \leq j$ implies that $time(\alpha_i) \leq time(\alpha_j)$.

An *input sequence* is a timed-ordered sequence of external events. An *output sequence* is a timed-ordered sequence of internal events α_i such that $value(\alpha_i) \neq \perp$.

The *input sequence projection* $\overrightarrow{\alpha}|_{in}$ of an event trace $\overrightarrow{\alpha}$ is the sequence of those events of $\overrightarrow{\alpha}$ that are external events, preserving the same order in which they appear in $\overrightarrow{\alpha}$. Similarly, the *output sequence projection* $\overrightarrow{\alpha}|_{out}$ of $\overrightarrow{\alpha}$ is its projection onto internal events whose output value is not $\perp$.

An *execution with respect to an input sequence* $\overrightarrow{\alpha}$ or *run* is an execution with event trace $\overrightarrow{\beta}$ such that $\overrightarrow{\beta}|_{in} = \overrightarrow{\alpha}$. Define the set of all runs of M with input $\overrightarrow{\alpha}$ as

$$run(M, \overrightarrow{\alpha}, init) \stackrel{def}{=} \{\overrightarrow{\gamma} | \overrightarrow{\gamma} \in executions(M, init) \text{ and } trace(\overrightarrow{\gamma})|_{in} = \overrightarrow{\alpha}\}$$

4.2.3. *Compositionality.* As mentioned in section 2, compositionality is a fundamental property of a formalism to support a modular approach to modelling, analysis and simulation. Our main contribution at this point is the establishment of compositionality for the semantics provided above with respect to strong bisimilarity. Here we reproduce the definition of strong bisimilarity [**?**, 20, 21]:

Definition 4.7. Given an LTS $(S, A, \rightarrow)$, a *simulation* is a binary relation $R \subseteq S \times S$ such that if $(s_1, s_2) \in R$ then whenever $s_1 \xrightarrow{\alpha} s_1'$ for any $s_1' \in S$ and any $\alpha \in A$, there is an $s_2' \in S$ such that $s_2 \xrightarrow{\alpha} s_2'$ and $(s_1', s_2') \in R$. We say that s_1 is *simulated by* s_2, written $s_1 \preceq s_2$ if there is a simulation R such that $(s_1, s_2) \in R$. The relation $\preceq$ is called *similarity*. A *bisimulation* is a binary relation $R \subseteq S \times S$ such that R and R^{-1} are both simulations. We say that s_1 is *bisimilar* to s_2, written $s_1 \sim s_2$ if there is a bisimulation R such that $(s_1, s_2) \in R$. The relation $\sim$ is called *bisimilarity*.

Compositionality requires an equivalence relation to be preserved by all contexts. Hence, in order to establish compositionality we need to define what we mean by a *context*. We distinguish two related notions of context: *structural context*, and *state context*. Informally a structural context $C[\eta]$ is an underdefined coupled DEVS with a "hole" or placeholder η, which when replaced by an actual DEVS component A, will yield a fully-defined DEVS, denoted $C[\eta \rightarrow A]$ or simply $C[A]$. A state context, or *partial state* $\rho_C[\eta]$ of a structural context $C[\eta]$ is a mapping associating each subcomponent with a configuration (without associating the placeholder to a configuration.) We define an analogous notion of replacing the placeholder of a state context $\rho_C[\eta]$ by an actual fully-defined configuration (s, t), denoting it $\rho_C[\eta \rightarrow (s, t)]$. We distinguish between two kinds of contexts: *elementary*, where the placeholder is at depth 1, and *arbitrary*, where the place holder is at depth greater or equal to 1.

If A and B are two DEVS components, we write $A \downarrow (s, t) \sim B \downarrow (s', t')$ to mean that $(s, t) \sim (s', t')$ in the LTS which is the disjoint union of the LTSs for A and B[2]

We establish the following (for proofs, see [30]), which ensures that bisimilarity preserves the imminent sets.

Lemma 4.8. *Let $D = (X, Y, N, C, infl, Z, sel)$ be a coupled DEVS. If ρ_1 and ρ_2 are two D-states such that there is a $m_0 \in N$ for which $\rho_1(m_0) \sim \rho_2(m_0)$ and for all $m \in N$ such that $m \neq m_0$, $\rho_1(m) = \rho_2(m)$, then $imm_D(\rho_1) = imm_D(\rho_2)$.*

We say that two DEVS components are *mutually compatible* if they have the same input and output sets.

The following are the main results stating compositionality.

Theorem 4.9. *Let A and B be any mutually compatible DEVS components, and let $(s_A, t) \in Configs(A)$ and $(s_B, t) \in Configs(B)$ be any configurations. Given any elementary DEVS context $C[\eta]$ such that both A and B are compatible with $C[\eta]$, and given any partial state $\rho_C[\eta]$ of $C[\eta]$, if $A \downarrow (s_A, t) \sim B \downarrow (s_B, t)$ then $C[A] \downarrow (\rho_C[\eta \rightarrow (s_A, t)], t') \sim C[B] \downarrow (\rho_C[\eta \rightarrow (s_B, t)], t')$ for any t'.*

Corollary 4.10. *Let A and B be any mutually compatible DEVS components, and let $(s_A, t) \in Configs(A)$ and $(s_B, t) \in Configs(B)$ be any configurations. Given any arbitrary DEVS context $C[\eta]$ such that both A and B are compatible with $C[\eta]$, and given any partial state $\rho_C[\eta]$ of $C[\eta]$, if $A \downarrow (s_A, t) \sim B \downarrow (s_B, t)$ then $C[A] \downarrow (\rho_C[\eta \rightarrow (s_A, t)], t') \sim C[B] \downarrow (\rho_C[\eta \rightarrow (s_B, t)], t')$ for any t'.*

Conjecture 4.11. *$A \downarrow (s_A, t_A) \sim B \downarrow (s_B, t_B)$ if and only if $executions(A, (s_A, t_A)) = executions(A, (s_A, t_A))$*

[2]For full details see [30]

4.3. **Devslang.** Devslang is a language we have developed to describe DEVS systems which introduces a limited form of modality specifically tailored to aid modelling infinite-state systems.

A Devslang "program" consists of several "classes" of components. We have two types of components for atomic and coupled models respectively. Each component class specifies its input and output ports, and in addition it may be parametrized.

Atomic components are given by a set of "mode definitions." Each mode definition corresponds to a family of states which share the same internal structure and behaviour. It can be thought of as a function whose parameters represent the internal structure of the state (the variables,) and the behaviour is given by specifying external and internal transitions, time-advance and output. External transitions are specified using a form of pattern-matching.

A coupled component consists of a set of *component instantiations* which create individual components from their corresponding "class" by passing appropriate parameters. The coupled component also specifies the connections between component instances, and the selection function.

5. Sketch of the proposed formalism

In this section we present a general overview of the proposed formalism.

5.1. **Motivation.** Why do we need a new formalism to deal with variable-structure systems? The main reasons are summarized here:

- We want to be able to model complex and large variable-structure systems easily.
- We want to take advantage of the features of Statecharts and DEVS. These two formalisms not only address many issues of complex dynamic systems, but also enjoy a large user base.
- We want to overcome the limitations, problems and complexities of existing approaches, in particular we want to deal with time and (re)initialization of components.
- We want to be able to simulate as well as reason about such systems. Most existing approaches rarely provide the means to do both.

This means we need a formalism expressive enough to address these issues, yet simple enough to use, to analyze (manually or automatically, e.g. that supports model-checking techniques,) to run efficiently, and to support code generation.

5.2. **Informal presentation.** As mentioned before, "structure" can be described by some graphical structure such as plain graphs, hypergraphs or higraphs. The particular graphical structure used depends on the aspects of the systems which need to be represented. For instance, if the system consists of several processes or entities connected by channels, and these channels are unidirectional and can be shared between more than two processes, then the appropriate representation of this structure would be a directed hypergraph. Systems which are composite, are better represented by some variant of higraphs or bigraphs.

Behaviour of certain systems, in particular discrete-time and discrete-event systems, can also be described in a graphical way. This has been evident since some of the earliest formal descriptions of system behaviour: formalisms such as finite-state automata, Petri Nets and Statecharts are inherently graphical in nature. These however, provide only a very limited representation or none of the purely structural aspects of a system in a graphical fashion.

The formalism that we propose presents both structure and behaviour of (discrete) systems in the same graphical way.

At the core of the formalism we find a notion of hierarchical graph (higraph for short,) which must capture two central aspects of structure: connectivity between components, and nesting of components.

Higraphs have been widely used in existing formalisms to represent structure and behaviour, but, to our knowledge, existing formalisms allow the description of only one of these. For instance, Statecharts describe only the behavioural aspects (and a few, limited structural aspects,) while a graphical notation of the Ambient calculus describes on its own only the structure. In the latter case, the behaviour of the system is given by a state transition system which defines the operational semantics of the language.

A common approach to describe the operational semantics of a language is to give a state transition system. This however, is orthogonal to (separated from) the language itself. In most cases, such state

transition system could be seen as an "executive," (in a way similar to DS-DEVS) which directs the execution of a system. In the case of dynamic-structure systems, this "executive" chooses between the possible structures at each step in time. But state transition systems can be viewed in a graphical way on their own.

Another possible graphical description which combines both approaches is to give first the nesting and connectivity of a system, and then the behavioural aspects. In this approach, nesting, as usual, is given by a tree. And behaviour is given in the leaves of this tree only. A typical example of this would be a variant of the DEVS formalism.

The formalism that we propose is based on the notion that behaviour might be given, graphically, not only at the leaves of the nesting tree, but at any level, similar to Statecharts, and unlike Statecharts, connectivity of components is described explicitly. This means that we could visualize this as a crossbreed between Statecharts and purely structural Higraphs. So nodes (blobs) and (hyper)edges have different meaning in different nodes. In some, nodes represent states and edges represent transitions, while in others, nodes represent components and edges represent connections (channels of communication.)

The following are the most significant features besides the "mix" of structure and behaviour in the higraph.

- Structural blobs have typed-interfaces: an interface is a set of ports, each of which has an associated type. A component with an input port of type X can receive as input any signal whose type is a subtype of X.
- Communication is performed through message-passing on unidirectional channels.
- There are two types of transitions: external and internal. External transitions have an input and/or output associated. Internal, transitions have a timeout associated. External transitions take place due to interaction with other concurrent components at the same level of nesting. Internal transitions take place autonomously when the time has expired.
- A large family of states can be described by a "mode," which can be seen as a parametrized set. The mode is thus a class of states, which when the system is running, will be instantiated to a particular state. The "behavioural nodes" of the graph represent such modes, rather than specific states.

5.3. **Differences with Statecharts and DEVS.** The main difference with Statecharts is the treatment of concurrency and communication. In Statecharts, all communication is by global broadcasting. In our formalism, concurrent components communicate through specific channels.

Another significant difference is the treatment of initialization. In Statecharts, each composite state has a designated default state. In our formalism, the specification of such states is given separately.

The main difference with DEVS is that the behavioural components are not exclusively the leaves of the nesting tree.

5.4. **Generalized compositionality and generalized contexts.** Compositionality is linked to abstraction: if a semantics is compositional, all systems deemed to be equivalent are the same from the point of view af any observer, and therefore any environment or context can ignore internal details of individual systems.

Compositionality can be difficult to achieve, and it depends on the equivalence and the semantic properties considered. Furthermore, in some cases it might be too strong of a property. For example, as mentioned in section 2.5.5, we know from Dataflow Theory that the equivalence of input/output relation is not compositional for process networks with indeterminate primitives (see [25].) However, if we restrict those systems to determinate networks, compositionality is achieved (see [19].) Similarly, in the π-calculus it is known that the so-called *ground bisimilarity* is not a congruence, but if we restrict the systems to be *pure generators* (that is, processes that do not perform input,) then compositionality is recovered.

Thus far, we have cited the standard approach to compositionality, but is this the only link to abstraction? In this thesis we propose looking at the role of compositionality at different levels. To do this, we propose a generalization of the concept of context.

The traditional notion of context is that of a network (a multicomponent system, i.e. a coupling specification,) where at least one of the components is a "placeholder." The placeholder only specifies minimal structural requirements, namely, having a specific interface. Based on this, the notion of substituting the placeholder by a system (or a state) is defined.

Our generalization of contexts allows the placeholder to impose additional requirements, even behavioural requirements on the systems that might be "plugged-in."

We denote a specification A at the observation frame as A^{obs}, at the I/O relation frame as A^{ior}, at the I/O function frame as A^{iof}, at the state transition level as A^{st}, and at the coupled level as A^{net}. Notice that the levels form a partial order. In Ziegler's hierarchy, it actually is a total order where $obs < ior < iof < st < net$, but we could consider alternative hierarchies, for instance one where st and net are not comparable.

Then a context $C[A^x]$, called an A^x-*context*, is an element of $netctxt[x]$, the set of all networks which include a subcomponent A^x for some $x \neq net$ (and usually $x < net$), that has the role of a "placeholder." Then the notion of substitability is defined as long as we are substituting for something at least as concrete as the placeholder: $C[A^x \to B^y]$ denotes replacing A^x by B^y in C, but this can be defined only if $x \leq y$ and B^y can replace A^x. What is the meaning of "can replace?" It means that B^y must satisfy at least the requirements of A^x. For example, B^{st} can replace A^{ior} if the input/output relation generated by B^{st} is a superset of that given by A^{ior}. In general, if $x \neq y$, such notion of replaceability requires an abstraction morphism. If $x = y$, a morphism at that level (x) suffices.

This notion of context could be generalized further to deal with multiple placeholders.

Given such a general notion of context, we obtain a corresponding notion of compositionality. For instance, we could say that an equivalence R is A^x-compositional, where x is an abstraction level, if it is preserved by all A^x-contexts. Hence an A^x-compositional equivalence would relate only systems that satisfy the specification A^x.

It is in light of such notions that we propose to study the compositionality properties of the proposed formalism.

6. Thesis objectives and Contributions

As mentioned in the introduction, the main goal of this thesis is the development of a framework for modelling, simulating and analyzing systems with variable structure, and the study of existing formalisms for this purpose. The formalisms studied suggest common patterns used to model this kind of system. The proposed framework must satisfy the following criteria:

(1) It must be well-founded: there must be a formal semantics suitable to reason about systems.
(2) It must be expressive: it must capture the most common patterns of mobility and variable structure with relative ease.
(3) It must be practical: there must be elements in the framework that allow it to deal with real problems. In particular it must be "time-aware." Furthermore, there must be tool support for modelling and simulation, and possibly for verification.

In order to satisfy these criteria, we must investigate both the theoretical aspects of variable-structure modelling, and the implementation of modelling and simulation tools supporting the approach.

6.1. Theoretical aspects.

The study of the theoretical aspects encompasses several key issues.

- Operational semantics: we look into both the traditional approach to operational semantics (SOS) based on terms, as well as a graph-grammar-based approach, and the relation between the two approaches, in particular for DEVS, Statecharts, and the new formalism.
- Behavioural equivalence: we study interesting forms of behavioural equivalence, a notion fundamental to the understanding of the dynamics of a system. In the context of DEVS, we have proposed several variants of bisimulation, whose properties should be further explored.
- Compositionality: in order to support modular and incremental development, the syntax and the semantics of a language should be compositional. We have recently established the compositionality of strong bisimulation for DEVS systems. Compositionality for Statecharts has been established for certain subsets of the formalism. We seek a formalism where compositionality is achieved without too many restrictions on the syntax.

- Generalized contexts: in order to support generalized compositionality we need to develop the notion of generalized context as outlined in section 5.4. The properties of such contexts and their role in the hierarchy of system specification must be studied.
- Fully-abstract denotational semantics: to support reasoning at a more abstract level, a denotational semantics is desirable, but such semantics is useful only if is matches the operational semantics. Some preliminary investigation suggests that results from dataflow theory impose certain restrictions on the possible domains for such semantics for DEVS. These would be carried to any formalism based on DEVS.
- Behaviour-preserving formalism translations: to support a multi-formalism approach to modelling and simulation, we need translation mechanism which preserve the semantics of the formalisms. The techniques for establishing such results for inductively defined formalisms (both in syntax and semantics) are well understood. Such techniques, however, have not been fully explored in the context of meta-modelling and graph-grammars, where the sets of interest are not inductively defined. We need to investigate further the required techniques.
- Component reinitialization: a well-known problem in hybrid systems is that of reinitialization: after a structural (or mode) transition, what should be the state of the system? In the resulting mode, the previous state might be illegal, and a new state must be chosen. The formalisms discussed in section 3 brush this issue under the carpet, by fixing the target state (e.g. having a fixed default initial state.) This approach is however very inflexible in practice. Modellers wish to leave initialization of components separate from the specification of the component. The question then is to find the appropriate syntactic and semantic mechanisms to allow flexible initialization of (sub)components.

6.2. **Tool Support.** The empirical component of the thesis consists of the development of modelling, simulation and possibly analysis tools for the proposed formalism.

This implementation will be done by means of meta-modelling and *graph-grammars*. Metamodelling provides us with a simple approach to describe syntax of graphical formalisms, and graph-grammars gives us a way to both describe transformations between models and to specify operational semantics. By specifying a grammar in such a way we obtain a prototype simulator by means of a graph-grammar engine. We will use the tool AToM^3 (see [16]) which allows us to specify both meta-models and graph-grammars.

6.3. **Contributions.** Here we make explicit the contributions we have made so far, the proposed general contributions, and some additional topics that can further investigated.

6.3.1. *Contributions made so far.* At this point we have investigated and developed the following:

(1) An operational semantics for DEVS which is compositional with respect to strong bisimulation and independent of specific simulation algorithms. (See [30]. Soon to be submitted for publication.)
(2) A textual form of DEVS, called Devslang, which introduces a limited form of modality specifically tailored to aid modelling infinite-state systems. This includes an explicit operational semantics. We are currently working on its translation into DEVS, and on an interpreter/simulator.
(3) A graphical DEVS modelling environment, and a graph-grammar based code generator from DEVS models to produce model-specific simulators (see [29].)
(4) A graph-grammar-based operational semantics for Causal Block Diagrams (see [31].)

6.3.2. *General core contributions.* There are several contributions proposed in this thesis:

- A full specification of a new formalism to model reactive systems in a compositional manner which attempts to unify two different perspectives on variable structure, explicitly deal with time and (re)initialization of components. This will encompass:
  - A basic theory which addresses general properties of these systems, in particular compositionality and determinacy, and
  - Support tools for modelling and simulation
- The establishment of the expressive power of DEVS with respect to Statecharts.

- Graph-grammar semantics for DEVS, Statecharts and the new formalism.

References

[1] Martin Abadi and Luca Cardelli. *A Theory of Objects*. Springer-Verlag, 1996.

[2] Gul A. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, 1986.

[3] Gul A. Agha and Carl Hewitt. Concurrent programming using Actors. In *Object-Oriented Concurrent Programming*. MIT Press, 1987.

[4] Paul C. Attie and Nancy A. Lynch. Dynamic Input/Output Automata: a formal model for dynamic systems. Technical Report MIT-LCS-TR-902, MIT Laboratory for Computer Science, July 2003.

[5] Michael Barr and Charles Wells. *Category Theory for Computing Science*. Prentice-Hall, 1990.

[6] F. Barros, M. Mendes, and B. Zeigler. Variable DEVS — variable structure modeling formalism: An adaptive computer architecture application. 1994.

[7] Luca Cardelli and Andrew D. Gordon. Mobile ambients. In *Foundations of Software Science and Computation Structures: First International Conference, FOSSACS '98*. Springer-Verlag, 1998.

[8] A. Corradini, U. Montanari, F. Rossi, H. Ehrig, R. Heckel, and M. Löwe. Algebraic approaches to graph transformation I: Basic concepts and double pushout approach. In G. Rozenberg, editor, *Handbook of Graph Grammars and Computing by Graph transformation, Volume 1: Foundations*. World Scientific, 1997.

[9] R. Frances, A. Evans, K. Lano, and B. Rumpe. The UML as a formal modeling notation. In *Proceedings of the OOPSLA'97 Workshop on Object-Oriented Behavioral Semantics*, 1997.

[10] David Harel. Statecharts: A visual formalism for complex systems. *Sci. Comput. Programming*, 8, 1987.

[11] David Harel. On visual formalisms. *Communications of the ACM*, 31(5), 1988.

[12] David Harel and Hillel Kugler. The Rhapsody semantics of Statecharts (or, on the executable core of the UML). In *Integration of Software Specification Techniques for Applications in Engineering*, volume 3147 of *LNCS*. Springer-Verlag, 2004.

[13] David Harel and Amnon Naamad. The STATEMATE semantics of Statecharts. *ACM Transactions on Software Engineering and Methodology*, 5(4), 1996.

[14] Michael Huth and Mark Ryan. *Logic in Computer Science: Modelling and reasoning about systems*. Cambridge University Press, 1999.

[15] Dexter Kozen. Results on the propositional μ-calculus. In *ICALP '82*, 1982.

[16] Juan De Lara and Hans Vangheluwe. AToM^3: A tool for multi-formalism and meta-modelling. In *European Joint Conference on Theory And Practice of Software (ETAPS), Fundamental Approaches to Software Engineering (FASE)*, volume 2306 of *LNCS*. Springer-Verlag, 2002.

[17] Gerald Lüttgen and Michael Mendler. The intuitionism behind Statecharts steps. *ACM Transactions on Computational Logic*, 3(1), 2002.

[18] Gerald Lüttgen, Michael von der Beeck, and Rance Cleaveland. A compositional approach to Statecharts semantics. In *Proceedings of the 8th ACM SIGSOFT international symposium on Foundations of software engineering*. ACM Press, 2000.

[19] Nancy A. Lynch and Eugene W. Stark. A proof of the kahn principle for input/output automata. *Information and Computation*, 1989.

[20] Robin Milner. *A Calculus of Communicating Systems*. Springer-Verlag, 1980.

[21] Robin Milner. *Communication and Concurrency*. Prentice Hall, 1989.

[22] Robin Milner. Bigraphical Reactive Systems: basic theory. Technical Report UCAM-CL-TR-523, University of Cambridge, Computer Laboratory, September 2001.

[23] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, parts I and II. Reports ECS-LFCS-89-85 and 86, Computer Science Dept., University of Edinburgh, March 1989.

[24] Pieter J. Mosterman and Hans Vangheluwe. Computer Automated Multi-Paradigm Modeling. *ACM Transactions on Modeling and Computer Simulation*, 12(4), 2002.

[25] Prakash Panangaden. The expressive power of indeterminate primitives in asynchronous computation. In *Proceedings of FSTTCS*, 1995.

[26] David Park. Concurrency and automata on infinite sequences. *Theoretical Computer Science*, 1981.

[27] Gordon Plotkin. A structural approach to operational semantics. Lecture Notes DAIMI FN-19, Dept. of Computer Science, Aarhus University, 1981.

[28] Amir Pnueli and Michal Shalev. What is in a step: On the semantics of Statecharts. In *Theoretical Aspects of Computer Software*, volume 526 of *LNCS*. Springer-Verlag, 1991.

[29] Ernesto Posse. Generating DEVS modelling and simulation environments. In *Summer Computer Simulation Conference (SCSC'03), Student Workshop*, 2003.

[30] Ernesto Posse. A compositional operational semantics for DEVS components. Technical report, School of Computer Science, McGill University, 2004.

[31] Ernesto Posse, Juan De Lara, and Hand Vangheluwe. Processing causal block diagrams with graph-grammars in AToM^3. In *European Joint Conference on Theory and Practice of Software (ETAPS), Workshop on Applied Graph Transformation (AGT)*, 2002.

[32] Thomas A. Henzinger Pei-Hsin Ho Rajeev Alur, Costas Courcoubetis. Hybrid automata: An algorithmic approach to the specification and verification of hybrid systems. volume 736 of *LNCS*. Springer-Verlang, 1993.

[33] James Rumbaugh, Ivar Jacobson, and Grady Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, 1999.

[34] Davide Sangiorgi. *Expressing Mobility in Process Algebras: First-Order and Higher-Order Paradigms*. Ph.D. thesis, Department of Computer Science, University of Edinburgh, 1992.

[35] Andrew C. Uselton and Scott A. Smolka. A compositional semantics for Statecharts using labeled transition systems. In *Proceedings of CONCUR '94 – Fifth International Conference on Concurrency Theory*, LNCS. Springer-Verlag, 1994.

[36] Bernard P. Zeigler, Herbert Praehofer, and Tag Gon Kim. *Theory of Modeling and Simulation*. Academic Press, second edition, 2000.

Appendix: Plan

Here we outline the proposed time-table. We plan to divide the development in several phases as follows:

(1) Complete work in progress
(2) Fundamentals for the new framework
(3) Development of the formalism
(4) Modelling tools
(5) Analysis
(6) Simulation

In the first phase we plan to complete work in progress with respect to the study of properties of DEVS models, the development of Devslang and the translation from Statecharts into Devslang, including its proof of correctness.

In the second phase, we plan on focusing on some fundamental aspects required to develop the semantics of the proposed formalism. In particular, we will focus on the relations between Graph-Grammars and inductively-defined LTSs. Furthermore we will study in more depth the role of generalized contexts in the hierarchy of system specifications, and their properties.

In the third phase we will focus on developing the formalism itself.

In the fourth we will focus on modelling tools for the formalism.

In the fifth we will focus on the analysis component, more specifically on the establishment of compositionality and other basic properties of interest.

In the sixth we will focus on simulation.

Finally we will allocate some time for additional topics to pursue.

Below there is a table with the estimated times (in weeks) for these phases. The estimated times are based on previously done work and work in progress.

- Pending issues:

Topic	Subtopics	Estimated time
DEVS properties	Closure under coupling Determinacy Agreement of bisimilarity and execution equivalence.	4
Devslang	Semantics of pattern-matching Mapping into DEVS Interpreter	4 - 8
Statecharts-to-Devslang	Statechart semantics Translation Proof (simulation) Proof (FA) Implementation	4 - 8
Total		10 - 20 (2.5 - 5 months)

- General issues:

Topic	Subtopics	Estimated time
Graph-Grammars/LTS relation		2 - 4
Generalized contexts		4
Total		6 - 6 (1.5 - 2 months)

- New formalism:
  - Basics:

	Syntax	Semantics	Total
Notion of Structure	1 - 2	2 - 4	3 - 6
Notion of change	1 - 2	2 - 4	3 - 6
(Re)initialization	1 - 2	2 - 4	3 - 6
Time	1 - 2	2 - 4	3 - 6
Total			12 - 24 (3 - 6 months)

  - Modelling:

Topic	Estimated time
Textual syntax	(see basics)
Graphical syntax	(see basics)
Tool	4 - 8
Total	4 - 6 (1 - 2 months)

  - Analysis (Properties):

Topic	Subtopics	Estimated time
Compositionality		4 - 8
Determinacy		4 - 8
Relations (1st prior)	to Statecharts to DEVS to HSS	2 - 4 2 - 4 2 - 4
Relations (2nd prior)	to DS-DEVS to π, ambients to GG to BRS to Dataflow	2 - 4 2 - 4 2 - 4 2 - 4 2 - 4
(opt)Modal logic		4
(opt)Other properties		4 - 8
Total (1st prior)		14 - 28 (3.5 - 7 months)
Total (2nd prior)		32 - 60 (8 - 15 months)

  - Simulation (Algorithms):

Topic	Estimated time
Abstract algorithm	4
Implementation	4
(opt)Model checking	4
Total	8 - 12 (2 - 3 months)

- Total estimated time: 54 - 92 weeks (13.5 - 23 months)