Rialto 2.0: A Language for Heterogeneous Computations

Johan Lilius, Andreas Dahlin, Lionel Morel

▶ To cite this version:

Johan Lilius, Andreas Dahlin, Lionel Morel. Rialto 2.0: A Language for Heterogeneous Computations. Distributed, Parallel and Biologically Inspired Systems, Sep 2010, Brisbane, Australia. pp.7-18, 10.1007/978-3-642-15234-4_3 . hal-00521339

HAL Id: hal-00521339
https://hal.science/hal-00521339

Submitted on 27 Sep 2010

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Rialto 2.0: A Language for Heterogeneous Computations

Johan Lilius[1], Andreas Dahlin[1,2], and Lionel Morel[3]

[1] Center for Reliable Software Technology, Åbo Akademi University, Finland
[2] Turku Centre for Computer Science, Finland
{jolilius,andalin}@abo.fi
[3] Université de Lyon, France
lionel.morel@insa-lyon.fr

Abstract. Modern embedded systems are often heterogeneous in that their design requires several description paradigms, based on different models of computation and concurrency (MoCCs). In this paper we present Rialto, a formal language intended at expressing computations in several MoCCs. The distinguishing features of Rialto and its implementation are 1) A formal semantics: the language is formalized using SOS (structured operational semantics) rules; 2) Encapsulation of models of computation into policies: we thus distinguish between the syntactic elements of the language (parallelism, interrupts) and its semantics; 3) efficient implementation algorithms. Policies are expressed in the language itself, which allows for more expressive power and a sounder semantics.

1 Introduction

A model of computation (MoC) is a domain specific, often intuitive, understanding of how the computations in that domain are done: it encompasses the designer's notion of physical processes, or as Edward A. Lee [1] puts it, the "laws of physics" that govern component interactions. Many different computational models exist: Hardware is often seen as having a synchronous model of computation in the sense that everything is governed by a global clock, while software has an asynchronous MoC. A system that is described using several MoCs is called heterogeneous, and the computations it makes are *heterogeneous computations.*

We are interested in understanding what the combination of models of computation means. The need for combining several models of computation arises often when modelling embedded systems. Our specific interest is in understanding the combination of models of computation from an operational perspective. Figure 1 shows an example of a system modeled in two different models of computation: One of the states in a state machine is refined by a Synchronous dataflow (SDF) graph. While in state wait, the program can take a transition to state process and start processing events using the algorithm in the SDF diagram. However several questions need to be answered before this description can be implemented. For example: what happens if a second e1 arrives while the system is in state process?

M. Hinchey et al. (Eds.): DIPES/BICC 2010, IFIP AICT 329, pp. 7–18, 2010.
© IFIP International Federation for Information Processing 2010

Fig. 1. A state machine, with one state refined by an SDF graph

In practice one does not program in a model of computation but in a programming language and we have therefore taken a slightly broader definition and view a model of computation as consisting of both a language and a corresponding semantics. The goal of our research can now be stated as twofold: 1. The development of a unified operational mathematical model of models of computation, and 2. the development of a textual language, in which it will be possible to program these kinds of models using a standard set of syntactic elements to represent entities in the different models of computation.

The second goal is motivated by the fact that many of the languages we have looked at (e.g. UML state machines [2], Esterel [3] and Harel's Statecharts [4]), use the same syntactic concepts but with different semantics. What we would like to do is pinpoint the semantic differences to certain syntactic concepts. For example the notion of parallelism exists in all three languages above, but there is certainly a difference in the semantics of parallelism between UML state machines and Esterel. On the other hand all languages also have a notion of interrupt (the trap-construct in Esterel and hierarchical transitions in both variants of Statecharts) that have very similar semantics.

To address this issue, we propose a language for expressing computations in several models of computation. The distinguishing features of Rialto and its implementation are: 1. *A formal semantics*: The language is formalized using SOS rules, 2. *Encapsulation of models of computation into policies*: This technique makes it possible to distinguish between the syntactic elements of the language (like parallelism, interrupts) and its semantics (step, interleaving, rtc, etc.) and 3. *Efficient implementation algorithms*: A Rialto program can be flattened [5]. This means that there exists a path to an efficient implementation.

The paper is structured in the following way. In section 2, we describe syntax and motivate the choice of syntactic entities. In section 3 we briefly outline the operational semantics, and the scheduling semantics of the language. Finally in the last sections, we present some examples and give a conclusion.

1.1 Related Work

The work of Lee et al. [6, 7] is a comprehensive study of different models of computation. The authors propose a formal classification framework that makes it possible to compare and express differences between models of computation. The framework is denotational and has no operational content, which means that it is possible to describe models of computation, including timed and partial order based models that we cannot model in our framework. The reason for this is that both timed and partial order based models are models that describe

program	::= **program** *name decbody* **begin** *body* **end;**	*nullstmt*	::= **null**
decbody	::= (*vardec* | *owndec* | *policdec*)*	*body*	::= ((*label*:)? (*S*|*expr*);)*
vardec	::= (*label*:)? **var** *name*: *Type*;	*gotostmt*	::= **goto** *label*
policydec	::= **policy** *name decbody* **begin** *body* **end;**	*atomicstmt*	::= [*body*]
ifstmt	::= **if** *boolexpr* **then** *body* **else** *body* **endif**	*returnstmt*	::= **return** *label*
parstmt	::= **par** *body* || *body* **endpar**	*suspendstmt*	::= **suspend** *label*
statestmt	::= **state policy** *name*; *decbody* **begin** *body* **endstate**	*resumestmt*	::= **resume** *label*
		assignstmt	::= *i* **:=** *expr*
		trapstmt	::= **trap** *boolexpr* **do** *S* *body* **endtrap**

Fig. 2. The Rialto grammar (S represents any statement)

constraints on possible implementations. Although we can model dataflow in our language, we have to decide on a specific operational semantics for the dataflow. This semantics will be one of several that preserve the partial-ordering between operations described by the dataflow specification. On the other hand Girault et al. [8] present ideas for combining graphically modelled MoCs, e.g. they combine SDF graphs with finite state machines. Their idea is similar to ours in that they use state hierarchy to delineate MoCs.

We would also like to point out that in [6], Lee independently proposes an approach that is conceptually essentially the same as ours, i.e. he suggests that a language, or a set of languages, with a given abstract syntax, can be used to model very different things depending on the semantics and the model of computation connected to the syntax. More recently, Benveniste et al. have provided interesting insights on dealing with heterogeneity through so-called Tag Systems [9,10]. Their approach, which is also based on a denotational description of the possible traces of a system, provides a mathematical setting well suited for proving properties on the correctness of particular design methods. Our work, on the other hand, proposes a language for programming heterogeneous systems, letting the user designing both the hierarchical structure of the program and the scheduling policies that rule each sub-system. From a language point of view, Rialto is also close to Ptolemy [11]. Essentially, our states are Ptolemy's actors while our policies can be seen as formal descriptions of Ptolemy's directors. Central differences are that Rialto has a formal semantics and code generation, while Ptolemy is a modelling and simulation tool.

2 Syntax of the Language

In this section we define the syntax of Rialto 2.0, discuss the choice of syntactic elements and provide an example. Our language is a small language, originally designed to describe UML statecharts. Basic syntax is given in Figure 2. Each statement in a program has a unique label, given by a designer or the compiler.

The basic concept in our language is the notion of a *state*. State is seldom explicit in programming languages like VHDL or ESTEREL but many modelling

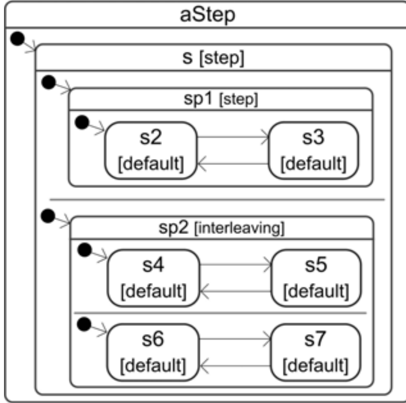
languages like UML, Harel's Statecharts or Petri nets make state explicit. Rialto states can be concurrent as well as hierarchical, sequential computations inside states can be expressed in a connected action language. Syntactically a state is represented by a state - endstate block.

An *interrupt* is an event of high priority that should be reacted upon immediately, or almost immediately. In our language, a trap - endtrap block is used to monitor interrupts. Interrupts correspond to trap in ESTEREL and hierarchical transitions going upwards in the state hierarchy in UML and Harel's Statecharts. *Coroutines* are independent threads of control that can be suspended and resumed. In programming languages, threads and processes are common abstraction mechanisms for coroutines. In modelling languages coroutines play a crucial role, e.g. history states in UML and Harel's Statecharts label the thread of control in a state as a coroutine, because the state is suspended when a hierarchical transition takes the control out of the state. In Rialto 2.0, *concurrency* is indicated using the `par` statement. The parallelism is interpreted differently depending on the execution policy for the current scope.

A novelty in our language is that we make *atomicity* explicit. Atomicity defines what the smallest observable state change is. At the one extreme, in traditional programming languages, atomicity is not a part of the language itself, but is loosely defined by multiprogramming concepts like semaphores and monitors. At the other extreme, in synchronous languages like Esterel, atomicity encompasses the whole program, so that the internal workings of the program are not observable. In the middle-field between these extremes other proposals exist, e.g. the GALS (Globally Asynchronous, Locally Synchronous) semantics proposed in POLIS [12]. In GALS atomicity is confined to single state machines, while communication between state machines can be observed. In our approach we have introduced atomicity as an explicit syntactic entity, the atomic brackets [ ]. It abides to the normal rules of scoping and is thus less general than the first approach mentioned above, but using this approach we can model its interaction with other constructs at the needed level of atomicity.

The *communication policy* states how different modules of the system communicate with each other. For the moment we have taken a rather simple approach which allows us to still model many more complex approaches. We call the main communication media in our language *channels*. A channel can e.g. represent the global event queue in a UML statechart, a link in an SDF graph etc. In state diagrams, an event is an occurrence that may trigger a state transition. In UML statecharts, there is an implicit global event queue; whereas, in our language several channels can be declared and the scope of a channel declaration is the state block. The notation in our language for checking for the presence of an event on a queue is *q1.e1*, where *q1* is the queue and *e1* is an event.

Data handling is not our primary concern at the moment, as we are more interested in control-dominated programming; however, the language has a few primitive types like integers and floats. Complex types and functions are only declared in Rialto, while their implementation is deferred to the target language. This is the same approach as in ESTEREL.




```
program aStep begin
 s: state
  policy step; begin
   s_par: par
    sp1: state policy step; begin
      s2: state policy default; begin
       s2_do:[print s2;goto s3;]; endstate;
      s3: state policy default; begin
       s3_do:[print s2;goto s3;]; endstate;
    endstate;
   ||
    sp2: state policy interleaving; begin
      sp2_par: par
       sp2_par_l: state policy default; begin
       ... endstate;
      ||
       sp2_par_r: state policy default; begin
       ... endstate;
      endpar;
    endstate;
   endpar;
 endstate;
end;
```

Fig. 3. A hierarchical state machine with two policies

2.1 Example

Figure 3 gives a graphical and textual representation of a simple Rialto program. This program encodes a hierarchical state machine composed of two machines `sp1` and `sp2` that are put in parallel with the || statement. The latter is itself decomposed into two state machines put in parallel, `sp2_par_l` and `sp2_par_r`. To each state in the program is associated a scheduling policy which defines how execution is organized within the state. The default, step and interleaving policies as well as other policies are also defined in Rialto. They are discussed in section 4.

In the initial state, the state machine will be in the states `s2`, `s4` and `s6`. Execution starts with the evaluation of the top-most policy shared by these three states, i.e. the policy `step` associated to `s`. This policy is defined (see section 4) to execute each orthogonal state. In this case both states `sp1` and `sp2` should be executed during the step. Our initial states are organized into partitions {`s2`} and {`s4, s6`}. The execution of `s2` in the first part of the step is straightforward and it will result in a transition to the state `s3`. The next part of the step is to let the policy associated with {`s4, s6`}, namely the `interleaving` policy, decide the execution of these states. The `interleaving` policy is defined to randomly execute one of the orthogonal regions of the state it is scheduling. This means that either `s4` or `s6` is executed, but never both during the same step. In the scenario where `s4` is executed the result of the interleaving would lead `sp2` into the state {`s5, s6`}, while the other scenario (execution of `s6`) would move `sp2` into {`s4, s7`}. The step is completed by collecting the new state of the system. The instance of the `step` policy, which is scheduling `s`, is responsible to collect the new observable state of the system. This state is either {`s3, s5, s6`} or {`s3, s4, s7`}. The next step will also be initiated by the policy of `s`, since the new state of the system also has `s` as its parent state.

3 Semantics

The semantics of Rialto 2.0 is split into three parts. First we define the static structure of a Rialto program. This is a graph that encodes the hierarchical and sequential relationships of statements in the program. Then we define the dynamic state of a Rialto program. Finally we explain the operational rules that are used to interpret a Rialto program.

3.1 The Static Structure of a Rialto Program

A Rialto program consists of a set of hierarchical state-machines. Each statement enclosed in a state-block has a label that acts as an "instruction address". Because of the hierarchical structure we can define a tree structure on these labels, which reflects the hierarchy of the program. There is a sequential order on some of the statements; reflected in the fact that leafs of a node in the tree may be ordered using a next relation. A program is defined as a tuple $<\mathcal{L}, \downarrow, \rightharpoonup, \mathcal{P}>$, where:

- $\mathcal{L}$ is the set of labels of the program. Labels are strings ($\mathcal{L} \subset Strings$);
- $\downarrow$ is a tree on $\mathcal{L}$;
- $\rightharpoonup$ is a partial function on $\mathcal{L}$ that defines the next relation between labels;
- $\mathcal{P}$ is the function **Label** $\rightarrow$ **Stmt**, that maps each label to a statement.

3.2 Dynamic State of a Rialto Program

The state of a Rialto program is a stack of state configurations. By default, the top element of the stack is always selected for execution. A *state configuration* is defined by $\mathcal{SC} = \mathcal{L}^* \times \mathcal{L}^*$, the set of pairs of lists of labels, representing state configurations. We have $\forall sc \in \mathcal{SC}. sc = (active, suspended)$ where: $sc.active \subseteq \mathcal{L}$ designates the set of active labels in the state configuration, while $sc.suspended \subseteq \mathcal{L}$ designates the set of suspended labels in the state configuration.

A state configuration is used to represent the dynamic state of a Rialto program, i.e. it basically contains the list of "sub-processes" that are either subject to execution (the *active* set) or that should not be executed because they have been suspended (the *suspended* set). Thus, suspend and resume actions of co-routines can easily be modelled by moving labels between the active and suspended sets.

A policy instance represents the dynamic state of execution of a particular scheduling policy associated to a particular state. It is defined as a tuple $\mathcal{P} = \mathcal{L} \times \mathcal{L} \times \mathcal{E}nv$. We have $\forall p \in \mathcal{P}.\ p = (callLabel, currentLabel, ownVars)$ where:

- $p.callLabel \in \mathcal{L}$ designates the label of the instance of the policy currently used (the label where the policy is "called").
- $p.currentlabel \in \mathcal{L}$ designates the label *in the policy*: the place in the policy where this instance of the policy is currently at.
- $p.ownVars \in \mathcal{E}nv$ designates the variable environments corresponding to *this* particular instance of the policy. Encodes the state of the policy.

We will use one such stack to organize the dynamic execution of a Rialto program. This stack is used to memorize execution context, in particular, when switching between user (program) and supervisor (policy) modes. Stack elements are tuples made of a *Cell.* $\mathcal{C} = \mathcal{SC} \times \mathcal{C} \times \mathcal{P}$ is the set of cells. We have: $\forall c \in \mathcal{C}.c = (sc, prevProgCtx, policyDesc)$ where:

- $c.sc \in \mathcal{SC}$ the current state configuration that gives information about suspended and active labels in the currently executing context,
- $c.policyDesc \in \mathcal{P}$ designates the current policy instance that is "leading" the execution of the current state configuration,
- $c.prevProgCtx \in \mathcal{C}$ designates the previous program context

We define *Stacks of state configurations.* $\mathcal{S}tack(\mathcal{SC})$ denotes the type "stack of $\mathcal{SC}$ elements". We denote $top(st).active$ as $st.active$, while $top(st).suspended$ is denoted $st.suspended$. The stack $\mathcal{SC}$ can be seen as an interleaving of the program and the policy execution stacks. As we have chosen to write policies in Rialto, it is natural to use the same stack structure to represent their state. This corresponds to the normal operating system states user and supervisor. But this means that we need special functions to distinguish between these two states.

Finally we can define the *runtime state of the program* $\mathcal{RS}tack : \mathcal{S}tack \times \mathcal{E}nv \times \mathcal{L}$ denotes the type "Rialto program stack". We have $\forall$ $rstack \in \mathcal{RS}tack = (st, env, pc)$ where st designates the program's stack, env designates the variable environment for the program and pc (program counter) is a pointer to the currently executed statement. This stack is at the heart of the semantics of the language. It is also available in the language itself. Indeed, it serves both for dealing with the basic language mechanisms (see section 3.3), and in the description of the scheduling policies.

3.3 Semantics of Statements

In this section we will discuss the operational rules for executing a Rialto program. We will assume the existence of the Rialto dynamic structures $r = (st, env, pc)$ as introduced earlier. Every statement has the same structure. The program counter points to a statement in the program array. The rule is selected by matching on this statement. There may be other conditions that have to be true. If the premise holds, then the rule is "executed". Finally the program counter is set to $\perp$ to force the control to the "enter policy" rule.

$$\frac{\mathcal{P}[\texttt{Pc}] = \text{“}stmt\text{''} \quad \text{“}otherconditions\text{''}}{\text{“}stmtstatechange\text{''} \quad \texttt{Pc} = \perp}$$

The `null` statement (0) deletes the current label from the active set and adds the successor. The `if` statement (1) is also very easily defined. We have two branches, the true and the false branch. The `par` statement (2) is a compound statement. The assumption is that all compound statements have their substatements as children in the label-tree. So the effect is to delete the label of the par-statement and to add all its children to the active set.

0	$\dfrac{\mathcal{P}l[\mathsf{Pc}] = \text{“}null'' \wedge \mathsf{Pc} \neq \bot}{st.\mathsf{active} = st.\mathsf{active} \setminus \{\mathsf{Pc}\} \cup \mathsf{next(Pc)} \wedge \mathsf{Pc} = \bot}$	6	$\dfrac{\mathcal{P}[\mathsf{Pc}] = \mathtt{trap}\ b\ \mathtt{do}\ stmt \wedge \mathsf{Pc} \neq \bot \wedge \mathsf{eval}(b, \mathtt{Var})}{st.\mathsf{active} = st.\mathsf{active} \setminus \{\mathsf{Pc}\} \cup \mathsf{label}(stmt) \wedge \mathsf{Pc} = \bot}$
1	$\dfrac{\mathcal{P}[\mathsf{Pc}] = \ \mathtt{if}\ b\ \mathtt{then}\ stmt_1\ \mathtt{else}\ stmt_2\ \mathtt{endif} \wedge pc \neq \bot \wedge \mathsf{eval}(b, \mathtt{Var})}{st.\mathsf{active} = st.\mathsf{active} \setminus \{\mathsf{Pc}\} \cup \mathsf{label}(stmt_1) \wedge pc = \bot}$	7	$\dfrac{\mathcal{P}[\mathsf{Pc}] = varname ::= expr;}{\mathtt{Var}[varname] = eval(expr, \mathtt{Var}) \wedge \mathsf{Pc} = \bot}$
	$\dfrac{\mathcal{P}[\mathsf{Pc}] = \ \mathtt{if}\ b\ \mathtt{then}\ stmt_1\ \mathtt{else}\ stmt_2\ \mathtt{endif} \wedge pc \neq \bot \wedge \neq \mathsf{eval}(b, \mathtt{Var})}{st.\mathsf{active} = st.\mathsf{active} \setminus \{\mathsf{Pc}\} \cup \mathsf{label}(stmt_2) \wedge pc = \bot}$	8	$\dfrac{\mathcal{P}[\mathsf{Pc}] = \ \mathtt{state} \wedge \mathsf{Pc} \neq \bot}{st.\mathsf{active} = st.\mathsf{active} \setminus \{\mathsf{Pc}\} \cup \mathsf{next(Pc)} \wedge \mathsf{Pc} = \bot}$
2	$\dfrac{\mathcal{P}[\mathsf{Pc}] = \ \mathtt{par}\ \ stmt(\Vert\ stmt)^*\ \ \mathtt{endpar} \wedge \mathsf{Pc} \neq \bot}{st.\mathsf{active} = st.\mathsf{active} \setminus \{\mathsf{Pc}\} \cup \mathsf{child(Pc)} \wedge \mathsf{Pc} = \bot}$	9	$\dfrac{\mathcal{P}[\mathsf{Pc}] = \ \mathtt{program} \wedge \mathsf{Pc} \neq \bot}{st.\mathsf{active} = st.\mathsf{active} \setminus \{\mathsf{Pc}\} \cup \mathsf{next(Pc)} \wedge \mathsf{Pc} = \bot}$
3	$\dfrac{\mathcal{P}[\mathsf{Pc}] = \ \mathsf{suspend}\ l \wedge \exists l' \in \mathsf{subtree}(l) : l' \in \mathsf{active}(st) \wedge \mathsf{Pc} \neq \bot}{st.\text{suspended} = st.\text{suspended} \cup \{st.\mathsf{active} \cap \mathsf{subtree}(l)\} \quad st.\mathsf{active} = st.\mathsf{active} \setminus \{\mathsf{Pc}, \mathsf{subtree}(l)\} \wedge \mathsf{Pc} = \bot}$	10	$\dfrac{\mathsf{Pc} = \bot}{\mathsf{Pc} = \mathsf{lub}(st.\mathsf{active}).policyDesc}$ $\mathsf{push}(st, new\mathcal{C}(\{\mathsf{Pc}\}, \mathsf{Pc}, \mathsf{top}(st)))$
4	$\dfrac{\mathcal{P}[\mathsf{Pc}] = \ \mathtt{resume}\ l \wedge l \in \mathsf{active}(st) \wedge \mathsf{Pc} \neq \bot}{st.\text{suspended} = st.\text{suspended} \setminus \{subtree(l)\} \quad st.\mathsf{active} = st.\mathsf{active} \setminus \{\mathsf{Pc}\} \cup \{st.\text{suspended} \cap \mathsf{subtree}(l)\} \wedge \mathsf{Pc} = \bot}$	11	$\dfrac{\mathsf{Pc} \neq \bot \wedge \mathcal{P}[pc] = \ \mathtt{return}\ l}{\mathsf{Pc} = env[l] \wedge \mathsf{pop}(st)}$
5	$\dfrac{\mathcal{P}[\mathsf{Pc}] = \ \mathsf{goto}\ \{l_1, \ldots, l_n\} \wedge \mathsf{Pc} \neq \bot}{st.\mathsf{active} = st.\mathsf{active} \setminus \{\mathsf{Pc}\} \cup \mathsf{subtree}(\mathsf{lub}(\mathsf{path}(\mathsf{Pc}, \mathsf{lub}(\mathsf{Pc}, l_1, .., l_n)))) \cup \{l_1, \ldots, l_n\} \wedge \mathsf{Pc} = \bot}$	12	$\dfrac{\mathsf{Pc} = \bot \wedge \mathsf{top}(st) = \varphi}{\mathsf{pop}(st)}$

Fig. 4. The statement (0-9) and policy rules (10-12)

The `suspend` (3) statement deletes a label from the active set and moves it into the set of suspended labels, effectively suspending the executing of the corresponding thread. There are two ways this statement can be defined. The first and the more simple one is to assume that for the statement to make sense, the label l must be active. On the other hand, when writing a scheduling policy for an operating system, it might make sense to be able to suspend a task without knowing which statement it was executing at the time. For this reason we choose a definition for `suspend` that actually suspends the subtree below it. However if this subtree is not active then the command is a “nop”. `resume` (4) is the companion to `suspend`. It moves a label from the suspended set into the active set thus resuming the thread. As with the `suspend` statement we have to take care that the whole subtree is resumed.

A goto (5) statement should jump control from the current location to the location pointed to by the label l. For this we need to calculate the least upper bound between the goto statements label and l. Then we delete all children of lub(l, `Pc`) that are on the path to `Pc` from the active set and add all the children that are on the path to l. The `trap` (6) statement is a statement that monitors a certain condition. Anytime it is executed it checks the condition. If the condition holds the do part is executed, else nothing is done. In both cases the trap statement is reactivated. Note that for the trap statement to be effective, it should be executed at each step. However, no such execution is built into the Rialto language. Instead this has to be taken care of by the policy.

The *assignment* statement is defined in rule (7). The expression is evaluated in the environment and the resulting value is assigned to the variable. The state declaration (8) is used to delineate a hierarchical state. The nature of a state is such that the program will stay in the state until it is exited from the state explicitly,

through a goto statement or by other means. Thus the execution of a `state` statement just adds the label of the first statement in the state block to the active set, while the `endstate` statement restarts the state block. Finally a `program` statement is used to start the execution of the program.

Policy Protocol. In Figure 4 the policy rules are shown. A policy controls what a step in the execution of the program consists of. It has three possible states:

1. the *initialisation*, at which point the contents of the step is calculated and the first label is selected,
2. the *execution* state, in which the policy selects the next label from a set of labels calculated in the initialisation state and
3. the *exit* state, in which the policy manipulates the stack and returns.

We define the *entry to a policy* (10) as follows. A policy execution can only start if the value of the program counter is the special label $\perp$. Every rule that wants to trigger the execution of a policy must set the program counter to this special value at the end of its execution. Then we pick the top element from the state configuration stack, and find the least upper bound of this set. The label is then one whose policy we will start executing. Notice that it is not enough to select a label and then pick its policy. The state may be spread out in several hierarchical states with different policies, thus we need to pick the policy of the lowest upper bound in this hierarchy to get at the right policy. We assign the address of this policy to the program counter. Finally we add a new state configuration with only the current value of the program counter. In effect this will confine the execution to the statements of the policy.

Exiting a policy (11) is done by executing a **return** statement. The *policy return protocol* requires that the policy always returns one label, which is the next label to be executed. This label, stored in variable l, is retrieved from memory by the rule. The top of the stack is now the last label of the policy, which means that the stack must be manipulated so that this label is replaced with the next label. Finally we restore the previous *state context*.

The last rule (12) presented in Figure 4 is necessary for dealing with the special case when every active label in the current state configuration has been executed. Then we need to pop the state configuration stack to get new labels to execute. If the stack is empty the program terminates.

Some policies require "non-destructive" evaluation of statements. This situation arises e.g. in the RTC-step of UML-state-machines [2], where the RTC-step first collects all "enabled" transitions, i.e. those transitions that can be executed, because their action is on the input queue. Then this set is pruned by deleting transitions whose guard is not true, or who are disabled by some other transition higher up in the hierarchy. For this we define a function $\mathsf{enabled} : \mathbf{Label} \times \mathbf{Var} \rightarrow \mathbf{Bool}$ that returns true if the statement attached to the label can be executed. Given the set of statements as defined above, all statements are by definition enabled all the time, except the **if** statement. For the latter, we define: $\mathsf{enabled}(\texttt{ if } b \texttt{ then } stmt_1 \texttt{ else } stmt_2 \texttt{ endif}) = \mathsf{eval}(b, \texttt{Var})$. An assumption here is that evaluation does not have any side-effects.

```
policy default
 own indefault: Boolean; var l: Label;
 begin
   l := sc.prevProgCtx.getLabelFromActiveSet();
   if indefault == true then
     indefault := false; sc.bottom().getActiveSet().add(l);
     if sc.size() > 2 then sc.popFromPrevProgCtx(); else endif;
     return __;
   else indefault := true && !sc.inPolicyMode(); return l; endif;
end;
```

Fig. 5. The default policy

4 Policies and Examples of Models of Computation

Using Rialto, the programmer is free to program the scheduling policies, i.e. models of computations. We now illustrate the description of such policies through several examples. Due to space reasons listings of all the policies cannot be presented, but they are available in [13]. The **default policy** (Figure 5) is used for completely sequential executions. As the name suggests, the policy is used as the default choice of policy for states. Scheduling decisions cannot be made by this policy, implying that it should only be used in situations where only one label is in the active labels set for the topmost stack element, i.e. when the next statement that can be executed is unique.

The **interleaving policy** is a loose, non-deterministic execution model. For example, UML Interactions (found in communication diagrams) can be scheduled by the interleaving policy. Each time it is activated, it selects randomly *one* label among the current active labels. In Figure 6, a code listing for this policy is provided. The policy is structured according to the policy protocol in the parts: *interleaving_init*, *interleaving_exec* and *interleaving_exit*. The first part of the policy contains the necessary variable declarations and decides which part of the policy should be executed, depending on the state of the particular instance of the policy (see Figure 6a). If the policy is already in the execution step we proceed to the *interleaving_exit* part presented in Figure 6c, but if the policy is activated in the beginning of an execution step the policy enters the initialisation state. In this state, all activated labels are collected (`calculateStep(currentPc)`) and a random active label is chosen for execution. The policy will now proceed to the execution state *interleaving_exec* (Figure 6b), in which necessary modifications of stack configurations are done and the label to be executed is put on top of the stack. The execution state is always completed by a return statement; either the label to execute is returned, implying that the program counter will be set to the returned label or the $\perp$ is returned, which indicates that another policy still must be invoked to decide on which statement is to be executed. In Figure 6c, the new state of the system is collected in the *exit* part of the policy. The new system state is made observable to the system by modifying the stack to reflect the new system state. Finally, the execution step is completed by returning $\perp$.

The **step policy** is used when we want to allow the computation to proceed in steps. A statement is executed in each concurrent thread at each step. The step policy is suitable to use in situations where real parallelism should be allowed,

```
policy interleaving
 own instep   : Boolean;
 own stackSize: Integer;
 own lbl      : Label;
 var lfound   : Boolean;
 var step:FifoQueueOfLabel
 var l        : Label;
 var runLbl : Label;
 var set    : SetOfLabel;
 var rLbls  : SetOfLabel;
 var scf    : StateConfig;
 var lblStr : String;

begin
 if instep then
   goto interleaving_exit;
 else
   goto interleaving_init;
 endif;

interleaving_init:
 instep := true;
 rLbls :=sc.prevProgCtx
  .getActiveSet();
 runLbl:=sc.prevProgCtx
  .getAnyActiveSetLbl();
 step :=calculateStep(PC);
 sc.prevProgCtx
  .getActiveSet().clear();
 lbl := step.poll();
 sc.prevProgCtx
  .getActiveSet().add(lbl)
 step.poll();
```

```
interleaving_exec:
 if step.empty() == false then
  lblStr := step.poll();
  if lblStr == __ then
   if lfound == true then
    scf.getActiveSet().add(set);
    sc.pushAbovePrevProgCtx(scf)
    scf.clear();
    rLbls.remove(set);
    lfound := false;
   else
    set.clear();
    goto interleaving_exec;
   endif;
  else
   l := lblStr;
   set.add(l);
   lfound := (lfound==false
    && l==runLbl)||lfound==true;
   goto interleaving_exec;
  endif;
 else endif;

 scf.getActiveSet().add(rLbls);
 sc.pushToBottom(scf);
 stackSize := sc.size();

 if lbl == runLblthen
   return runLbl;
 else
   return __;
 endif;
```

```
interleaving_exit:
 instep := false;
 scf:=
  sc.popFromPrevProgCtx();
 scf.getActiveSet()
  .remove(lbl);
 set := scf.getActiveSet();
 sc.bottom().getActiveSet()
  .add(set);

 if stackSize-sc.size()==1
 then
  sc.popFromPrevProgCtx();
 else endif;

 if sc.size() > 2 then
  scf :=sc.popFromBottom();
  set:=scf.getActiveSet();
  sc.bottom().getActiveSet()
   .add(set);
 else endif;

 return __;
end;
```

a) initialization **b) execution** **c) exit**

Fig. 6. Interleaving policy structured according to the three policy states

regardless of the chosen MoC. The policy can be seen, to some extent, as a replacement for the interleaving policy.

The **SDF policy** implements a policy for handling static dataflow. Although SDF is an abbreviation for synchronous dataflow, its underlying model is not synchronous so it can rather be described as an untimed MoC [5]. Synchronous dataflow is a special case of dataflow that requires scheduling decisions for the system can be taken already at compile time.

5 Conclusion and Future Work

We have presented Rialto, a uniform framework dedicated to the design of heterogeneous systems, based on the notion of model of computation. A MoCCs can be encoded in Rialto by writing a dedicated policy. Programs are structured using a state-based, which state being interpreted with respect to a policy that is associated to it. We have outlined several scheduling policies that are described more precisely in [13]. The latter also introduces JRialto, which is an interpreter for Rialto. Policies have been encoded and tested using JRialto.

This work can be continued in several ways. The first improvement that we are planning is to develop better abstractions for the stack manipulation. As can be seen in Figure 6 quite a lot of the code is actually housekeeping code for the stack. Better abstractions will make the writing of polices simpler and less

error-prone. The main reason for the complexity is the interleaving of the policy and program contexts on the stack. A second planned extension of the work is to implement the Rialto 2.0 semantics in HOL or some other proof assistant, to be able to prove properties of programs. Finally we will need to compare Rialto with other formalisms, among those presented in section 1.1. In particular, we would like to propose Rialto as an operational implementation of the Tag Systems [10].

References

1. Lee, E.A.: Embedded software. In: Zelkowitz, M. (ed.) Advances in Computers, vol. 56. Academic Press, London (2002)
2. Lilius, J., Paltor, I.P.: Formalising UML state machines for model checking. In: France, R.B., Rumpe, B. (eds.) UML 1999. LNCS, vol. 1723, pp. 430–445. Springer, Heidelberg (1999)
3. Berry, G., Gonthier, G.: The Esterel synchronous programming language: Design, semantics, implementation. Science of Computer Programming 19(2) (1992)
4. Harel, D.: Statecharts: A visual formalism for complex systems. Science of Computer Programming 8, 231–274 (1987)
5. Bjórklund, D.: A Kernel Language for Unified Code Synthesis. PhD thesis, Åbo Akademi University (2005)
6. Lee, E.A., Sangiovanni-Vincentelli, A.: A framework for comparing models of computation. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 17(12), 1217–1229 (1997)
7. Liu, X.: Semantic Foundation of the Tagged Signal Model. PhD thesis, EECS Department, University of California, Berkeley (2005)
8. Girault, A., Lee, B., Lee, E.A.: Hierarchical finite state machines with multiple concurrency models. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 18(6) (June 1999)
9. Benveniste, A., Caillaud, B., Carloni, L., Sangiovanni-Vincentelli, A.: Heterogeneous reactive systems modeling: Capturing causality and the correctness of loosely time-triggered architectures (ltta). In: Proc of the Fourth Intl. Conference on Embedded Software, EMSOFT. ACM, New York (2004)
10. Benveniste, A., Caillaud, B., Carloni, L.P., Caspi, P., Sangiovanni-Vincentelli, A.L.: Composing heterogeneous reactive systems. ACM Transactions on Embedded Computing Systems, TECS (2007)
11. Eker, J., Janneck, J.W., Lee, E.A., Liu, J., Liu, X., Ludvig, J., Neuendorffer, S., Sachs, S., Xiong, Y.: Taming heterogeneity — the ptolemy approach. In: Proceedings of the IEEE, pp. 127–144 (2003)
12. Balarin, F., Giusto, P., Jurecska, A., Passerone, C., Sentovich, E., Tabbara, B., Chiodo, M., Hsieh, H., Lavagno, L., Sangiovanni-Vincentelli, A.L., Suzuki, K.: Hardware-Software Co-Design of Embedded Systems, The POLIS Approach. Kluwer Academic Publishers, Dordrecht (1997)
13. Dahlin, A.: JRialto, an implementation of the heterogeneous Rialto modelling language. Master's thesis, Åbo Akademi University (2007), `http://www.abo.fi/~andalin/mastersthesis.pdf`