Stepwise Development of Simulink Models Using the Refinement Calculus Framework*

Pontus Boström[1], Lionel Morel[2], and Marina Waldén[1]

[1] Åbo Akademi University, Department of Information Technologies
Turku Centre for Computer Science
Joukahaisenkatu 3-5, 20520 Turku, Finland
{pontus.bostrom,marina.walden}@abo.fi
[2] INRIA/IRISA - Campus universitaire de Beaulieu
35042 Rennes Cedex, France
lionel.morel@irisa.fr

Abstract. Simulink is a popular tool for model-based development of control systems. However, due to the complexity caused by the increasing demand for sophisticated controllers, validation of Simulink models is becoming a more difficult task. To ensure correctness and reliability of large models, it is important to be able to reason about model parts and their interactions. This paper provides a definition of contracts and refinement using the action system formalism. Contracts enable abstract specifications of model parts, while refinement offers a framework to reason about correctness of implementation of contracts, as well as composition of model parts. An example is provided to illustrate system development using contracts and refinement.

1 Introduction

Simulink / Stateflow [16] is a domain specific programming and simulation language that has become popular for development of control- and signal-processing systems. It enables model-based design of control systems, where a (continuous) model of the plant can be constructed together with the (discrete) controller. Simulink offers a wide range of simulation tools, which enables simulation and evaluation of the performance of the controller. However, it lacks good tools and development methodologies for reasoning about correctness of models. In particular, it fails to enforce a structured stepwise development method that becomes necessary when developing large control systems.

A goal of our work is to establish such a development method by studying the application of formal analysis techniques to help validate models. The work is based on the use of assume-guarantee (called pre-post in this paper) contracts as a form of local specifications. Analysis techniques rely on a notion of refinement of Simulink models. The refinement calculus [6] gives a good theoretical

* This work is carried out in the context of the project ITCEE (Improving Transient Control and Energy Efficiency by digital hydraulics) funded by the Finnish funding agency TEKES

C.B. Jones, Z. Liu, J. Woodcock (Eds.): ICTAC 2007, LNCS 4711, pp. 79–93, 2007.
© Springer-Verlag Berlin Heidelberg 2007

framework for developing formal stepwise design methodologies in which abstract specifications are refined into detailed ones. The advantage with refinement is that it allows for a *seamless design flow* where every step in the development process can be formally validated by comparing the refined model to the more abstract one. The final implementation is then formally guaranteed to exhibit the behaviour described by the original specification.

Design-by-contract for embedded controllers. Contract-based design [18] is a popular software design technique in object-oriented programming. It is based on the idea that every method in each object is accompanied by (executable) pre- and post- conditions. The approach has also been successfully applied to reactive programs in [15]. There, a contract is described as a pair of monitors (A, G) and is associated to each component. The meaning of such a contract is that "*as long as input values satisfy A, the outputs should satisfy G*".

Contracts in Simulink consists of such pre- and post-conditions for model fragments [8]. To get a formal semantics of contracts we translate the Simulink models to action systems. Action systems [5,7] is a formalism based on the refinement calculus [6] for reasoning about reactive systems. Contracts are here viewed as non-deterministic abstract specifications. They cannot be simulated in Simulink, but they can be analysed by other tools e.g. theorem provers. Conformance of an implementation to a specification can be validated by model checking or testing. The aim to provide an easy to use and lightweight reasoning framework for correctness of models. Contracts together with the refinement definition also enable compositional reasoning [1] about correctness of models.

Other formalisations of Simulink exist [3,10,11,21,22]. Each one of these take into account different subsets of Simulink. Refinement of Simulink diagrams has also been considered by Cavalcanti et al. [10]. However, they deal mostly with refinement of models into code. We are interested in refinement and stepwise development of models from abstract specifications, which is not the concern of any of those works. Instead of action systems, a definition of refinement [19] for Lustre [13] could also be used in conjunction with the translation from Simulink to Lustre [22]. However, that formalisation can only accommodate discrete models. Treating continuous models using refinement of continuous action systems [17] is a rather natural extension of our formalisation. Furthermore, general formal description and refinement rules for process nets (block-diagrams) has also been investigated [14]. However, we focus specifically on rules for Simulink.

Here we only consider Simulink models that are discrete and use only one single sampling time. We do not consider all types of blocks, e.g., non-virtual subsystems or Stateflow. The action system formalism and refinement calculus is, however, very versatile [6] and can accommodate these features as well.

2 Action Systems

Action systems [4,5,7] are used for describing reactive and distributed systems. The formalism was invented by Back and Kurki-Sounio and later versions of it use the refinement calculus [6].

Table 1. Program statements with their predicate transformer semantics

$\langle f \rangle.q.\sigma$	$= q.f.\sigma$	(Functional update)
$\{p\}.q$	$= p \wedge q$	(Assertion)
$[p].q$	$= p \Rightarrow q$	(Assumption)
$(S_1; S_2).q$	$= S_1.(S_2.q)$	(Sequential composition)
$[R].q.\sigma$	$= \forall \sigma'.(R.\sigma.\sigma' \Rightarrow q.\sigma')$	(Demonic relational update)
$(S_1 \sqcap S_2).q$	$= S_1.q \wedge S_2.q$	(Demonic choice)
skip.q	$= q$	(Skip)
abort.q	$=$ false	(Aborted execution)
magic.q	$=$ true	(Miraculous execution)

Before introducing action systems, a short introduction to the refinement calculus [6] is needed. The refinement calculus is based on Higher Order Logic (HOL) and lattice theory. The state-space of a program in the refinement calculus is assumed to be of type Σ. Predicates are functions from the state-space to the type Boolean, $p : \Sigma \rightarrow bool$. A predicate corresponds to the subset of Σ where p evaluates to true. Relations can be thought of as functions from elements to set of elements, $R : \Sigma \rightarrow (\Sigma \rightarrow bool)$. A program statement is a predicate transformer from predicates on the output state-space Σ to predicates on the input state-space Γ, $S : (\Sigma \rightarrow bool) \rightarrow (\Gamma \rightarrow bool)$. Here we will only consider conjunctive predicate transformers [6,7]. Note also that conjunctivity implies monotonicity. A list of conjunctive predicate transformers are given in Table 1.The functional update consists of assignments of the type $(\langle f \rangle \mathrel{\hat{=}} x := e)$, where the value of variable x in state-space σ is replaced by e. The relational update R is given in the form $R \mathrel{\hat{=}} (x := x' | P.x.x')$. The predicate P gives the relation between the old values of variable x in σ and the new values x' in σ'. Other variables than x in σ remains unchanged in the updated state-space σ'.

An action system has the form:

$$\mathcal{A} \mathrel{\hat{=}} |[\ \textbf{var}\ x;\ \textbf{init}\ A_0;\ \textbf{do}\ A\ \textbf{od}\ ]| : \langle z \rangle$$

Here x (resp. z) denotes the local (resp. global) variables. A_0 is a predicate giving the initialisation action. All actions can be grouped as one single action A that consists of conjunctive predicate transformers, without loss of generality [7].

2.1 Trace Semantics

The execution of an action system gives rise to a sequence of states, called behaviours [5,7]. Behaviours can be finite or infinite. Finite behaviours can be aborted or miraculous, since we do not consider action systems that can terminate normally. In order to only consider infinite behaviours, aborted or miraculous behaviours are extended with infinite sequences of $\bot$ or $\top$, respectively. These states are referred to as *improper states*.

Action A can be written as $\{tA\}; [nA]$, where tA is a predicate and nA is a relation that relates the old and the new state-spaces. This can again be done without loss of generality [7]. Then $\sigma = \sigma_0, \sigma_1, \ldots$ is a possible behaviour of $\mathcal{A}$ if the following conditions hold [5,7]:

- The initial state satisfies the initialisation predicate, $A_0.\sigma_0$
- if σ_i is improper then σ_{i+1} is improper
- if σ_i is proper then either:

- the action system aborts, $\neg tA.\sigma_i$ and $\sigma_{i+1} = \bot$, or
- it behaves miraculously, $tA.\sigma_i \wedge (nA.\sigma_i = \varnothing)$ and $\sigma_{i+1} = \top$, or
- it executes normally, $tA.\sigma_i \wedge nA.\sigma_i.\sigma_{i+1}$

Behaviours contain local variables that cannot be observed. Only a trace of a behaviour consisting of global variables can be observed. Furthermore, all finite stuttering has been removed from the result and finally all infinite stuttering (internal divergence) has been replaced with an infinite sequence of $\bot$. Stuttering refers to steps where the global variables are left unchanged. The semantics of action system $\mathcal{A}$ is now a set of observable traces [5,7].

2.2 Refinement

Refinement of an action system $\mathcal{A}$ means replacing it by another system that is indistinguishable from $\mathcal{A}$ by the environment [5,7]. An ordering of traces σ and τ, $\sigma \leq \tau$, is first defined on the extended state-space $\Sigma \cup \{\bot, \top\}$ as:

$$(\sigma_0, \sigma_1, \ldots) \leq (\tau_0, \tau_1, \ldots) \mathrel{\hat{=}} (\forall i \cdot \sigma_i = \bot \vee \sigma_i = \tau_i \vee \tau_i = \top)$$

Consider two action systems $\mathcal{A}$ and $\mathcal{A}'$. Refinement is then defined as:

$$\mathcal{A} \sqsubseteq \mathcal{A}' \mathrel{\hat{=}} (\forall \sigma' \in tr(\mathcal{A}') \Rightarrow (\exists \sigma \in tr(\mathcal{A}) \cdot \sigma \leq \sigma'))$$

This means that for each trace in the refined system $\mathcal{A}'$ there exists a corresponding trace in the abstract system $\mathcal{A}$. Data refinement rules corresponding to this refinement semantics can be found in [5,7].

3 Encoding Simulink Diagrams in the Refinement Calculus

The structure of a Simulink *block diagram* can be described as a set of *blocks* containing *ports*, where ports are related by *signals*. Simulink has a large library of different blocks for mathematical and logical functions, blocks for modelling discrete and continuous systems, as well as blocks for structuring diagrams. Simulink diagrams can be hierarchical, where subsystem blocks are used for structuring.

3.1 The Steam Boiler Example

To illustrate the formalisation and stepwise development of Simulink, we use a simplified version of the steam boiler case study [2] as a running example throughout the paper. This system consists of a boiler for producing steam. Water is delivered to the system using a pump that can be switched on or off. Steam is taken from the boiler using a valve that is either fully opened or fully closed. The goal of the controller is to maintain the water level between the lower limit L_1 and upper limit L_2. It can read the current water level using the sensor *w_level* and it controls both the pump and the out-valve of the boiler through the actuators *pump_on* and *out_open*.

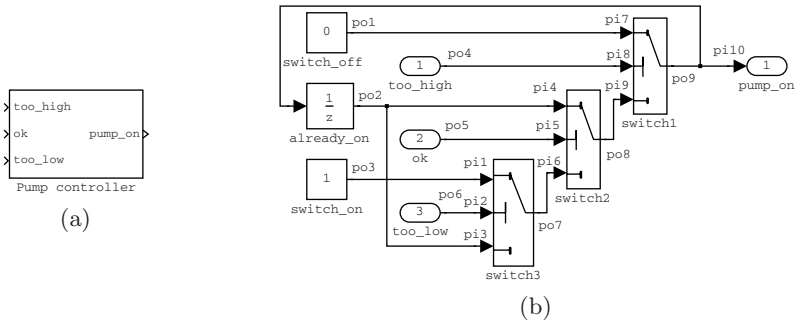

Fig. 1. A simple Simulink model consisting of a part from the steam boiler example. Diagram (b) shows the content of subsystem (a).

3.2 Translating Simulink Model Elements

To introduce the Simulink diagram notation, a small example is shown in Fig. 1. The diagram contains the part of the steam boiler controller that controls if the pump is switched on or off. The diagram contains source blocks *switch_on* and *switch_off* for defining constants giving the state of the pump. There are three in-blocks that provides Booleans stating whether the water level is *too high*, *ok* or *too low*. If the water level is too low the pump is switched on and if it is too high the pump is switched off by the switch-blocks, *switch1*, *switch2* and *switch3*. If the water level is between the limits, the current state of the pump is maintained using the memory in the block *already_on*. The desired state of the pump is delivered from the subsystem by the out block *pump_on*.

A Simulink model is defined as a tuple $\mathcal{M} = (B, root, \mathsf{subh}, P, \mathsf{blk}, \mathsf{ndep}, C)$.

- B is the set of blocks in the model. We can distinguish between the following types of blocks; subsystem blocks B^s, in-blocks in subsystems B^i, out-blocks in subsystems B^o and blocks with memory B^{mem}. When referring to other types of "basic" blocks, B^b is used in this paper. Subsystems B^s, in-blocks B^i and out-blocks B^o are referred to as *virtual blocks*, since they are used purely for structuring and have no effect on the behavioural semantics.
- $root \in B^s$ is the root subsystem.
- $\mathsf{subh} : B \to B^s$ is a function that describes the subsystem hierarchy. For every block b, $\mathsf{subh}.b$ gives the subsystem b is in;
- P is the set of ports for input and output of data to and from blocks. The ports P^i is the set of in-ports and P^o is the set of out-ports, $P = P^i \cup P^o$;
- $\mathsf{blk} : P \to B$ is a relation that maps every port to the block it belongs to;
- $\mathsf{ndep} : P^i \nrightarrow P^o$ is a partial function that maps in-ports in non-virtual blocks to the ports in other non-virtual blocks they depend on. An in-port depends on an out-port, if there is a signal or sequence of ports in virtual blocks connected by signals between them. Since we need to be able to analyse model fragments, all in-ports in non-virtual blocks are not necessarily connected. This function can be defined in terms of signals and ports, but for brevity it is given directly.

Table 2. Overview of the translation from Simulink to refinement calculus

Simulink construct	Requirements	Refinement calculus translation
Port, p:	$p \in P \wedge$ $\mathsf{blk}.p \notin (B^i \wedge B^o \wedge B^s)$	$\nu.p$
Constant, c:	$true$	c
Dependency, ndep:	$p_1 = \mathsf{ndep}.p_2$	$\nu.p_2 := \nu.p_1$
Normal block, b:	$b \in B^b \wedge \mathsf{blk}.p^o = b \wedge p^o \in P^o \wedge$ $\mathsf{blk}.p^i = b \wedge p^i \in P^i$	$\nu.p^o := f_b.(\nu.p^i).c_b$
Memory block, b:	$b \in B^{mem} \wedge \mathsf{blk}.p^o = b \wedge p^o \in P^o \wedge$ $\mathsf{blk}.p^i_f = b \wedge p^i_f \in P^i \wedge$ $\mathsf{blk}.p^i_g = b \wedge p^i_g \in P^i$	$\nu.p^o := f_b.p^i_f.x_b.c_b$ $x_b := g_b.p^i_g.x_b.c_b$

– C is the set of block parameters of the model. The block parameters are a set of constants defined in the Matlab workspace of the model.

There are several constraints concerning these functions and relations in order to only consider valid Simulink models, e.g., valid hierarchy of subsystems. In this paper we assume that we only deal with syntactically correct Simulink models (ones that can be drawn).

Consider the Simulink block diagram given in Fig. 1. The blocks are defined as $B \mathrel{\hat{=}} \{root, Pump\ controller, too\_high, ok, too\_low, switch1, \ldots\}$. The subsystems are given as $B^s \mathrel{\hat{=}} \{Pump\ controller, root\}$ and the hierarchy as $\mathsf{subh} \mathrel{\hat{=}} \{(Pump\ controller, root), (too\_high, Pump\ controller), \ldots\}$. Names of ports are usually not shown in diagrams. Here we have the following ports, $P \mathrel{\hat{=}} \{pi1, \ldots, pi10\} \cup \{po1, \ldots, po9\}$. The function describing to which block each port belongs to is then given as $\mathsf{blk} \mathrel{\hat{=}} \{(po1, switch\_off), (po2, already\_on), (po3, switch\_on), \ldots\}$. The connections between the ports are defined as $\mathsf{ndep} \mathrel{\hat{=}} \{(pi1, po3), (pi3, po2), (pi4, po2), \ldots\}$. Note that e.g. $(pi2, po6)$ is not in ndep, since $pi2$ is only connected to a virtual block. There are no configurable parameters C in this diagram.

To reason about Simulink models in the refinement calculus framework, all Simulink constructs are mapped to a corresponding construct in the refinement calculus, as shown in Table 2. The column *requirements* gives the required condition for a construct to be translated, while the column *refinement calculus translation* gives the actual translation.

Ports in Simulink corresponds to variables in the refinement calculus framework. The function $\nu : P \nrightarrow V$ describes this mapping, where V is a set of variable names. Only necessary ports are translated to variables, i.e., ports that can be in $(\mathrm{dom}\ .\mathsf{ndep} \cup \mathrm{ran}\ .\mathsf{ndep})$. The constant block parameters are translated directly to variables c. The connections between blocks, $p_1 = \mathsf{ndep}.p_2$, are modelled as assignments. A block can contain in-ports, out-ports, and block parameters. Each block $b \in B^b$ is associated with a function f_b that updates its out-ports based on the value of the in-ports p^i and the parameters of the block c_b. In blocks that contain memory $b \in B^{mem}$, the value on the out-ports depends also on the memory in the block x_b. The memory is updated (using a function g_b). These functions do not need to depend on all in-ports.

3.3 Ordering the Assignments Obtained from Simulink

Now that the translation of the individual constructs have been determined we can give the order in which to execute them. There are several different orderings, since diagrams can have blocks that do not depend on each other. To find an ordering, the dependency between ports in the Simulink diagram need to be determined. We define a relation totdep that describes this.

$$\begin{aligned}\mathsf{totdep} \mathrel{\hat{=}} \quad & \lambda p_1 : P \cdot \{p_2 \in P \mid \\ & ((p_1 \in P^i \Rightarrow p_2 \in \mathsf{ndep}.p_1) \wedge \\ & (p_1 \in P^o \Rightarrow p_2 \in \mathsf{fdep}.p_1))\}\end{aligned}$$

The relation totdep considers both the relation between ports as given by the signals and subsystem hierarchy (ndep), as well as the relations between out-ports and in-ports inside blocks (fdep). The relation fdep, $\mathsf{fdep} : P^o \to \mathcal{P}(P^i)$, can sometimes not be determined syntactically on the graphical part of the Simulink diagram. However, the data dependency for different blocks is documented in the Simulink documentation [16]. The relation totdep need to be a partial order that forms a directed acyclic graph for *deterministic models*. Hence, we can always find an order in which to update the ports in the model and ensure predictable behaviour and execution time. This is automatically ensured by Simulink, if the check for algebraic loops is activated. The order in which the translated Simulink model elements are executed can now be defined.

Definition 1 (Ordering of assignments). *Consider two ports p_1 and p_2 such that p_1 depends on p_2, $p_2 \in \mathsf{totdep}^*.p_1$. In the refinement calculus representation $\nu.p_1$ is updated in the substitution S_1 and $\nu.p_2$ in S_2. Then there exists a (possibly empty) sequence of substitutions S such that $S_2; S; S_1$.*

The ordering given in Def. 1 can be achieved by topologically sorting the assignments to ports. Note that this ensures that a port is never read before it has been updated.

Consider again the model in Fig. 1. The data dependency inside blocks is given by $\mathsf{fdep} \mathrel{\hat{=}} \{(po7, pi1), (po7, pi2), (po7, pi3), (po8, pi4), \ldots\}$, since the output of a switch block depends on all its inputs and the output of the memory block *already_on* does not depend on its input. The complete ordering of ports is then, $\mathsf{totdep} \mathrel{\hat{=}} \mathsf{ndep} \cup \mathsf{fdep}$. The refinement calculus representation $\mathsf{refCalc}.M$ of model M becomes:

$$\begin{aligned}\mathsf{refCalc}.M \mathrel{\hat{=}} \ & \nu.po1 := 0; \nu.pi7 := \nu.po1; \nu.po2 := x; \nu.pi3 := \nu.po2; \\ & \nu.po3 := 1; \nu.pi1 := \nu.po3; \nu.po7 := (\textbf{if } \nu.pi2 \textbf{ then } \nu.pi1 \textbf{ else } \nu.pi3 \textbf{ end}); \ldots \\ & x := \nu.po9\end{aligned}$$

Here x denotes the memory in block *already_on*. The memories are updated after the ports in the diagram have been updated. Hence, $\mathsf{refCalc}.M$ returns the sequential composition of a permutation satisfying the ordering rules of the individual translated statements, as well as the memory updates. Note that all in-ports are not assigned, since some in-ports are not connected to non-virtual blocks.

Table 3. Refinement calculus semantics of contract conditions

Contract condition	Refinement calculus semantics
$Q^{param}(c)$	$Q^{param}(c)$
$Q^{pre}(p^i, c)$	$\{Q^{pre}(\nu.p^i, c)\}$
$Q^{post}(p^o, p^i, c)$	$[\nu.p^o := v_o \mid Q^{post}(v_o, \nu.p^i, c)]$

4 Specification of Simulink Models

When developing Simulink models, we want to start with an abstract overview of the system that is then refined in a stepwise manner. We use contracts to give this abstract description.

The blocks in a Simulink diagram usually use parameters from the Matlab workspace. These parameters are required to have certain properties, here described by the predicate Q^{param}. In the refinement calculus this translates (see 3) into a condition describing the valid parameter values.

A contract for a Simulink model fragment consists of a pre-condition and a post-condition that state properties about its inputs and outputs. In practise this means that we give a *specification block* that can then be refined to a desired implementation. A specification block, M_s, contains in-ports (p^i), out-ports (p^o), a pre-condition (Q^{pre}) and a post-condition (Q^{post}). The semantics of the specification M_s is given by its translation to the refinement calculus shown in Table 3. Statements with this semantics cannot be simulated by the solvers in Simulink. However, other tools can be used to analyse these abstract specifications. The fact that an implementation satisfies its specification can be tested also in Simulink.

Consider again the steam boiler example. An overview of the complete system is given in Fig. 2. This model consists of a specification of the controller, *Controller*, and a specification of the plant, *Steam boiler*. The model has block parameters giving the maximum and minimum water levels L_1 and L_2, respectively. Water levels are positive and the upper level L_2 is higher than the lower level L_1, $Q^{param} \mathrel{\hat{=}} L_1 > 0 \wedge L_2 > L_1$. The following safety requirements are then given for the water level in the controller:

- When it is above L_2, the pump is switched off and the out valve is opened.
- When it is below L_1, the pump is switched on and the valve is closed.

The contract of the controller is then derived from the safety requirements.

$$
\begin{aligned}
&Q_c^{pre} \mathrel{\hat{=}} true \\
&Q_c^{post} \mathrel{\hat{=}} (w\_level > L_2 \Rightarrow \neg pump\_on \wedge out\_open) \wedge \\
&\qquad\qquad (w\_level < L_1 \Rightarrow pump\_on \wedge \neg out\_open)
\end{aligned}
$$

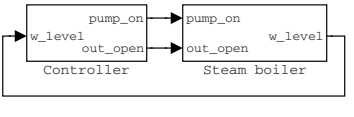

Fig. 2. The diagram in gives an overview of the complete steam boiler system

The plant has no pre-condition, $Q_p^{pre} \mathrel{\hat{=}} true$, and it can assign any positive value to the current water level, $Q_p^{post} \mathrel{\hat{=}} w\_level \geq 0$.

Since the ordering rules for statements in Def. 1 only concerns statements that updates variables, the assert statement needs a separate rule.

Definition 2 (Ordering of assert statements). *Consider an arbitrary assert statement* $\{Q(p_1, \ldots, p_n)\}$*. The assert statement is evaluated as soon as possible. This means that all statements* S_j *that updates* $\nu.p_j$*, where* $p_j \in \{p_1, \ldots, p_n\}$*, have already been evaluated. Furthermore, the last such update statement* S *is directly followed by the assert statement,* $S; \{Q(p_1, \ldots, p_n)\}$*.*

The diagram in Fig. 2 contains cyclic dependencies between blocks, i.e., feedback. Since this is a common pattern for connections between specification blocks, we also deal with it here. A cycle can be treated as a fix-point [14].

Definition 3 (Cyclic dependencies). *Assume that* M_s *is any specification block in a cycle. Let the in-port* p_s^i *of* M_s *be connected to the out-port* p^o *of another block in the same cycle. In order to be able to use the ordering rules in Definitions 1 and 2, this connection is considered broken when ordering statements. The refinement calculus translation of* M_s *gives the statements* $([\nu.p_s^i := v|true], [\nu.p_s^o := v|Q^{post}])$*. The rest of the constructs in the cycle are translated as in Tables 2 and 3. These statements are then followed by the assumption that the value of* p_s^i *and the value of* p^o *are equal and by the translated pre-condition* Q^{pre} *of the specification,* $[\nu.p_s^i = \nu.p^o]; \{Q^{pre}\}$*.*

The treatment of feedback here is similar to the one by Mahony [14]. It enables us to prove that the pre-condition is guaranteed by its preceding blocks. Using this technique, the diagram in Fig. 2 can now be translated:

$$\begin{array}{l}\mathsf{refCalc}.System \mathrel{\hat{=}} \\ \quad [w\_level' := v|true]; [pump\_on, out\_open := v_p, v_o|Q_c^{post}]; pump\_on' := pump\_on; \\ out\_open' := out\_open; \{Q_p^{pre}\};\ [w\_level := v|Q_p^{post}]; [w\_level' = w\_level]; \{Q_c^{pre}\}\end{array}$$

Cycles are not allowed in the implementation and new features have to be added during the refinement process to make the cyclic dependencies disappear.

4.1 Action System Semantics of Simulink Models

We have now given the semantics of all the needed parts of Simulink in the refinement calculus framework. The behaviour of the complete diagram is given as an action system. Assume that the constructs in the Simulink model is translated to the refinement calculus statements $(S_1, \ldots, S_n)$. This involves both standard Simulink constructs and contract statements. However, the execution order of these statements given in Def. 1-3 is not unique. Consider two arbitrarily chosen execution orders $S_k; \ldots; S_l$ and $S_r; \ldots; S_s$ satisfying the ordering constraints. The following results are then possible:

1 $(S_k; \ldots; S_l).false \wedge \neg(S_r; \ldots; S_s).true$
2 $\neg(S_k; \ldots; S_l).true \wedge (S_r; \ldots; S_s).false$
3 $\forall q \cdot (S_k; \ldots; S_l).q = (S_r; \ldots; S_s).q$

Due to the different order of statements, one sequence of statements might execute a miraculous statement before an abort statement or vice-versa (cases 1 and 2). Otherwise, the result is the same for both orderings (case 3).

Since a model should be non-terminating both miraculous and aborting behaviour are considered erroneous and should be avoided. Hence, we can consideran arbitrary ordering of the statements. The action system is then:

$$\begin{array}{l}\mathcal{M} \mathrel{\hat{=}} |[\ \textbf{var}\ x_1, \ldots, x_m, c; \\ \quad \textbf{init}\ Q^{param}(c) \wedge Init(x_1, \ldots, x_m); \\ \quad \textbf{do} \\ \quad\quad S_k; \ldots; S_l; R_1; \ldots; R_m; t := t + t_s \\ \quad \textbf{od} \\ ]| : \langle \nu.p_1, \ldots, \nu.p_n, t \rangle\end{array}$$

The global variables giving the observable behaviour are given by the ports, $p_1, \ldots, p_n$, in the model. This way it is possible to track that the behaviour of the initial model is preserved. The time t is considered to be a global variable, to ensure that we have no infinite stuttering. The memory of the blocks, $x_1, \ldots, x_m$, and constant block parameters c are local variables to the action system. The action consists of a sequence of statements $S_k; \ldots; S_l$ satisfying the ordering rules in Def. 1-3 that updates ports. This sequence is followed by statements $R_1; \ldots; R_m$ updating the memory variables $x_1, \ldots, x_m$. The order is not important, since these statements are deterministic and independent of each other. The system is correct, if all pre- and post-conditions are satisfied at all times. Correctness can be verified by checking that the system is non-terminating, $\forall t \cdot t \in tr(\mathcal{M}^V) \Rightarrow t \notin \{\bot, \top\}$.

4.2 Correctness of Simulink Models

The aim of this paper is to be able to define and verify correctness properties of Simulink models. Furthermore, since proofs might not always be feasible, we like to be able to have correctness criteria that can be model checked or tested.

Assume that we have a Simulink model M with a pre-condition Q^{pre} that should maintain a condition Q^{post}. Assume further that p_f^i denotes in-ports that are free and p_b^i denotes in-ports in Q^{pre} that are already connected. The translation of constructs of M with the pre-condition Q^{pre} and post-condition Q^{post} is given by the refinement calculus statements $(S_1, \ldots, S_n, \{Q^{pre}\}, \{Q^{post}\}, R_1, \ldots, R_m)$:

$$\mathsf{refCalc}.M = S_k; \ldots; S_j; \ldots; \{Q^{post}\}; \ldots; S_l; R_1; \ldots; R_m\ , \quad \text{for } k, j, l \in 1..n$$

The assert $\{Q^{pre}\}$ cannot be included, since not all in-ports are connected. Model M is therefore a *partial model*. However, we are interested in the model behaviour in the environment where it is used, i.e., for inputs, where the pre-condition holds. We create a *validation model* for obtaining a complete model that provides the most general environment of such type.

Definition 4 (Validation model). *A validation model is created by adding a non-deterministic assignment to the model that assigns the free in-ports p_f^i in*

M values satisfying the precondition. The model contains the refinement calculus statements: $(S_1, \ldots, S_n, \{Q^{pre}\}, \{Q^{post}\}, [\nu.p^i_f := v|Q^{pre}], R_1, \ldots, R_m)$. *The validation model for M is given as*

$$\mathsf{refCalc}.M^V \;\hat{=}\; S_k; \ldots; [\nu.p^i_f := v|Q^{pre}]; \ldots; \{Q^{pre}\}; \ldots; S_j; \ldots; \{Q^{post}\}; \ldots; S_l; R_1; \ldots; R_m$$

The behaviour of the validation model $\mathsf{refCalc}.M^V$ is given as an action system. The model M is correct, if the action system $\mathcal{M}^V$ has no improper traces.

If the model M is deterministic, the correctness of the validation model can be checked using model checking, other verification tools or testing. A test case for M is a model where the statement $[\nu.p^i_f := v|Q^{pre}(v, \nu.p^i_b)]$ has been refined to a deterministic statement. Note that we need to show that there is always a test case in order to ensure that the validation model does not behave miraculously.

If the model M is given as a set of specification blocks $M_1, \ldots, M_m$, where all the models M_j consists of a pre-condition Q^{pre}_j and post-condition Q^{post}_j, the correctness constraints can be simplified. In this case, there is no need to iterate the system over time, since the execution of the graph is independent of the number of times it has been executed before (see Def. 1-3). This lead to compositional reasoning about correctness for different model fragments similar to composition of specifications in [1], i.e., we do not have to know anything about the implementation of the specifications to prove that the connections between them are correct. We need to verify that 1) the validation model does not behave miraculously, $\neg M^V.false$ and that 2) the pre-conditions are not violated, $M^V.true$.

We can then derive a condition of the following type for checking pre-conditions in the model M^V using the refinement calculus:

$$([Q^{param}(c)]; [\nu.p^i_f := v|Q^{pre}]; \{Q^{pre}_1\}; [\nu.p^o_1 := v|Q^{post}_1]; \ldots; \\ \{Q^{pre}_m\}; [\nu.p^o_m := v|Q^{post}_m]; \{Q^{post}\}).true$$

Hence, the post-conditions of the predecessors have to imply the pre-condition of the successors and the final post-condition Q^{post}. Using weakest precondition calculations simple conditions can be derived. To show the absence of miraculous behaviour is similar.

5 Refinement

To get a definition of refinement we use the translation of Simulink to the Action Systems formalism. The abstract specifications given by contracts are refined to more concrete models. The properties of the block parameters were stated using an initialisation condition. Refinement of the block parameters follow standard rules for refinement of initialisation [5,7] and is not discussed here.

Consider specification block M_s with in-ports P^i_s, out-ports P^o_s, pre-condition Q^{pre} and post-condition Q^{post} in a model $M = (B, root, \mathsf{subh}, P, \mathsf{blk}, \mathsf{ndep}, C)$. This specification is refined by the model fragment $M_n = (B_n, root_n, \mathsf{subh}_n, P_n, \mathsf{blk}_n, \mathsf{ndep}_n, C_n)$ with pre-condition Q^{pre}_n. The refinement is illustrated in Fig. 3.

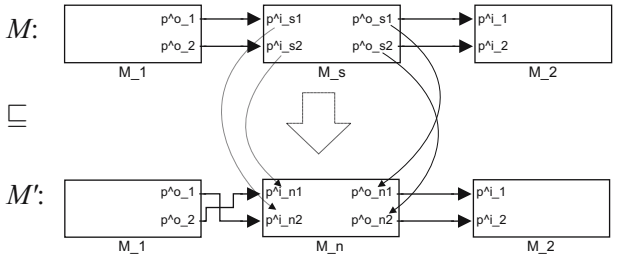

Fig. 3. Illustration of refinement of abstract specification M into refinement M'

The specification M_s is replaced by M_n, while the ports P_s^i and P_s^o of M_s are replaced by ports from M_n.

First we need to determine how ports in the old model M relates to ports in the new model M', in order to relate the variables $\mathsf{refCalc}.M$ to the variables in $\mathsf{refCalc}.M'$. The mapping of ports to variables is denoted by ν in the abstract model and by ν' in the refined model.

1. Every block except M_s in M is preserved in M'. Hence, each port p from these blocks are also preserved, which means that they are mapped to the same variables in the refinement calculus representation, $\nu'.p = \nu.p$.
2. For every in-port p from M_s in M there is an in-port p_n from M_n in M' such that they depend on the same port, $\mathsf{ndep}.p = \mathsf{ndep}'.p_n$ (see Fig. 3). The port p_n that replaces port p is mapped to the same variable in the refinement calculus representation, $\nu'.p_n = \nu.p$.
3. For every in-port p such that it depends on an out-port p_s in the specification block M_s there is a corresponding port p_n in M_n that p depends on, $p_s = \mathsf{ndep}.p \ \wedge \ p_n = \mathsf{ndep}'.p$ (see Fig. 3). The port p_n that replaces port p_s is mapped to the same variable as before in the refinement calculus representation, $\nu'.p_n = \nu.p_s$.

We need to show that the replacement of M_s with pre-condition Q_n^{pre} and model fragment M_n is a correct refinement. First we note that we can add an assert statement $\{Q^{post}\}$ after the statement $[\nu.p_{s1}^o, \ldots, \nu.p_{sn}^o := v|Q^{post}]$ in the abstract specification. In the refinement, the contract statements $(\{Q^{pre}\}, [\nu.p_{s1}^o, \ldots, \nu.p_{sn}^o := v|Q^{post}])$ are replaced by the translated Simulink model constructs $(\{Q_n^{pre}\}, S_1, \ldots, S_m, R_1, \ldots, R_t)$ obtained from M_n. We use a validation model to check the correctness of this refinement. This validation model uses the refinement calculus statements $([\nu.p_{s1}^i, \ldots, \nu.p_{sm}^i := v|Q^{pre}], \{Q^{post}\}, \{Q_n^{pre}\}, S_1, \ldots, S_m, R_1, \ldots, R_t)$. This model, $\mathsf{refCalc}.M_n^V$, is constructed of the statements above ordered according to the rules in Def. 1-3. Note that statement $[\nu.p_{s1}^i, \ldots, \nu.p_{sm}^i := v|Q^{pre}]$ assign the in-ports and, hence, appears in the beginning of the translation. The assert statement $\{Q^{post}\}$ that depends on the out-ports is placed towards the end.

$$\mathsf{refCalc}.M_n^V \ \hat{=} \ S_k; [\nu.p_{s1}^i, \ldots, \nu.p_{sm}^i := v|Q^{pre}]; \ldots; \{Q_n^{pre}\}; \ldots; S_l; \{Q^{post}\}; R_1; \ldots; R_t$$

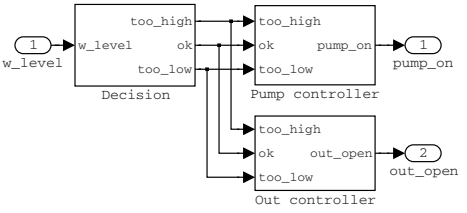

Fig. 4. This diagram shows the refinement of the controller

Theorem 1 (Correctness of refinement). *The model M_n refines M_s, $M_s \sqsubseteq M_n$, if $\forall t \cdot t \in tr(\mathcal{M}_n^V) \Rightarrow t \neq \bot$ and M_n does not behave miraculously.*

Proof. There are two constructs to consider $\{Q^{pre}\} \sqsubseteq \{Q_n^{pre}\}$ and $[\nu.p_{s1}^o, \ldots, \nu.p_{sn}^o := v|Q^{post}] \sqsubseteq \mathsf{refCalc}.M_n$.

- If $\{Q^{pre}\} \sqsubseteq \{Q_n^{pre}\}$ does not hold $\{Q_n^{pre}\}$ will contribute with aborted traces, due to the assignment to in-ports, $[\nu'.p_{s1}^i, \ldots, \nu'.p_{sm}^i := v|Q^{pre}]$.
- If $[\nu.p_{s1}^o, \ldots, \nu.p_{sn}^o := v|Q^{post}] \sqsubseteq \mathsf{refCalc}.M_n$ does not hold, then either,
  - the model fragment $\mathsf{refCalc}.M_n$ aborts, or
  - the output from $\mathsf{refCalc}.M_n$ does not satisfy Q^{post}.

  Both cases contribute with aborted traces.

Since we show that $\mathcal{M}_n^V$ does not abort we can conclude that $\{Q^{pre}\} \sqsubseteq \{Q_n^{pre}\}$ and $[\nu.p_{s1}^o, \ldots, \nu.p_{sn}^o := v|Q^{post}] \sqsubseteq \mathsf{refCalc}.M_n$ must hold. □

Due to monotonicity we have $M_s \sqsubseteq M_n \Rightarrow M \sqsubseteq M'$.

The controller in the steam boiler example is refined in a stepwise manner to obtain an implementation. Here we do the development in one single model, such that each level in the subsystem hierarchy is a new refinement step. Each subsystem is associated with a contract. When the system is refined, the details of the subsystem are added. First we refine the specification of the controller into three different subsystem as shown in Fig. 4. The first one, *Decision*, decides if the water level is too high (*too_high*), suitable (*ok*) or too low (*too_low*). The second one, *Pump Controller*, computes if the pump should be on, while the third one, *Out Controller*, computes if the out valve should be opened. The contract for the specification *Decision* states that the water level should be between L_1 and L_2 to be acceptable, otherwise it is too high or too low.

$$
\begin{aligned}
Q_d^{pre} &\triangleq true \\
Q_d^{post} &\triangleq (w\_level > L_2 \Rightarrow too\_high \wedge \neg ok \wedge \neg too\_low) \wedge \\
&\quad\ (w\_level < L_1 \Rightarrow \neg too\_high \wedge \neg ok \wedge too\_low) \wedge \\
&\quad\ (w\_level \geq L_1 \wedge w\_level \leq L_2 \Rightarrow \neg too\_low \wedge ok \wedge \neg too\_high)
\end{aligned}
$$

The specification block *Pump Controller* requires that the water level is either too low, acceptable or too high. It guarantees that the pump is switched on if the water level is too low and switched off if it is too high.

$$
\begin{aligned}
Q_{pump}^{pre} &\triangleq too\_high \vee ok \vee too\_low \\
Q_{pump}^{post} &\triangleq (too\_high \Rightarrow \neg pump\_on) \wedge (too\_low \Rightarrow pump\_on)
\end{aligned}
$$

The contract for the last specification, *Out Controller*, is defined similarly. Note that we do not say anything about the situation when the level is between L_1 and L_2. The implementation can choose the best alternative in that case.

To validate that the system in Fig. 4 refines the specification *Controller* we create a validation model:

$$\begin{array}{l}\mathsf{refCalc}.Controller^V \mathrel{\hat{=}} \\ \quad [w\_level := v | Q_c^{pre}(v)]; \{Q_d^{pre}\}; [too\_high, ok, too\_low := v_h, v_o, v_l | Q_d^{post}]; \\ \quad \{Q_{pump}^{pre}\}; [pump\_on := v | Q_{pump}^{post}]; \{Q_{out}^{pre}\}; [out\_open := v | Q_{out}^{post}]; \{Q_c^{post}\}\end{array}$$

We first need to show that the validation model does not behave miraculously, $\neg(\mathsf{refCalc}.Controller^V).false$. It is easy to see that values can always be given to the in-ports and that the post-conditions are feasible. The refinement is then correct if the validation model does not abort. By systematically computing the weakest precondition $(\mathsf{refCalc}.Controller^V).true$ from this program we get the following conditions:

$$\begin{array}{l}(Q^{param} \wedge Q_c^{pre} \Rightarrow Q_d^{pre}) \wedge \\ (Q^{param} \wedge Q_c^{pre} \wedge Q_d^{post} \Rightarrow Q_{pump}^{pre}) \wedge \\ (Q^{param} \wedge Q_c^{pre} \wedge Q_d^{post} \Rightarrow Q_{out}^{pre}) \wedge \\ (Q^{param} \wedge Q_c^{pre} \wedge Q_d^{post} \wedge Q_{pump}^{post} \wedge Q_{out}^{post} \Rightarrow Q_c^{post})\end{array}$$

The refinement of the *Controller* is still abstract and not executable. To illustrate the final implementation consider the implementation of the specification *Pump Controller* in Fig. 1. Here we have taken the approach to only switch on or off the pump when a water level limit is reached. Other control strategies can also be used. The implementation of *Pump Controller* uses memory and we have to validate its behaviour over time to ensure correct behaviour. This is again done by creating a validation model.

6 Conclusions and Future Work

In this paper we have presented a definition of refinement of Simulink diagrams using contracts and an action systems semantics. First we gave a translation from Simulink to refinement calculus and provided a definition of contracts using pre- and post-conditions in *specification blocks*. The action systems formalism provided semantics to these contracts. We then showed how an abstract specification given as a contract could be refined into an implementation satisfying the contract. Furthermore, validation of the refinement can be performed by model checking or testing a *validation model*. These ideas have been tried on a larger case study [8] and the initial experience with contracts and Simulink are positive. An extended version of the paper is also available as a technical report [9].

We believe this refinement-based development provides a convenient design method even for developers not familiar with formal methods. These methods are not limited to Simulink: they can be applied to other similar languages like SCADE [12] and Scicos [20] as well.

References

1. Abadi, M., Lamport, L.: Conjoining specifications. ACM Transactions on Programming Languages and Systems 17(3), 507–534 (1995)
2. Abrial, J.-R., Börger, E., Langmaack, H.: The steam boiler case study: Competition of formal program specification and development methods. In: Abrial, J.-R., Börger, E., Langmaack, H. (eds.) Formal Methods for Industrial Applications. LNCS, vol. 1165, pp. 1–12. Springer, Heidelberg (1996)
3. Arthan, R., Caseley, P., O'Halloran, C., Smith, A.: ClawZ: Control laws in Z. In: Proceedings of ICFEM 2000, pp. 169–176. IEEE Press, Los Alamitos (2000)
4. Back, R.-J.R., Kurki-Suonio, R.: Decentralization of process nets with centralized control. In: Proceedings of the 2nd ACM SIGACT-SIGOPS Symposium of Principles of Distributed Computing, pp. 131–142. ACM Press, New York (1983)
5. Back, R.-J.R., von Wright, J.: Trace refinement of action systems. In: Jonsson, B., Parrow, J. (eds.) CONCUR 1994. LNCS, vol. 836, pp. 367–384. Springer, Heidelberg (1994)
6. Back, R.-J.R., von Wright, J.: Refinement Calculus: A Systematic Introduction. Graduate Texts in Computer Science. Springer, Heidelberg (1998)
7. Back, R.-J.R., von Wright, J.: Compositional action system refinement. Formal Aspects of Computing 15, 103–117 (2003)
8. Boström, P., Linjama, M., Morel, L., Siivonen, L., Waldén, M.: Design and validation of digital controllers for hydraulics systems. In: The 10th Scandinavian International Conference on Fluid Power, Tampere, Finland (2007)
9. Boström, P., Morel, L., Waldén, M.: Stepwise development of Simulink models using the refinement calculus framework. Technical Report 821, TUCS (2007)
10. Cavalanti, A., Clayton, P., O'Halloran, C.: Control law diagrams in Circus. In: Fitzgerald, J.A., Hayes, I.J., Tarlecki, A. (eds.) FM 2005. LNCS, vol. 3582, pp. 253–268. Springer, Heidelberg (2005)
11. Chen, C., Dong, J.S.: Applying timed interval calculus to Simulink diagrams. In: Liu, Z., He, J. (eds.) ICFEM 2006. LNCS, vol. 4260, pp. 74–93. Springer, Heidelberg (2006)
12. Esterel Technologies. SCADE (2006), `http://www.esterel-technologies.com/`
13. Halbwachs, N., Caspi, P., Raymond, P., Pilaud, D.: The synchronous dataflow programming language lustre. Proceedings of the IEEE 79(9), 1305–1320 (1991)
14. Mahony, B.: The DOVE approach to design of complex dynamic processes. In: Theorem Proving in Higher Order Logic, NASA conf. publ., CP-2002-211736 (2002)
15. Maraninchi, F., Morel, L.: Logical-time contracts for reactive embedded components. In: ECBSE'04. 30th EUROMICRO Conference on Component-Based Software Engineering Track, Rennes, France (2004)
16. Mathworks Inc., Simulink/Stateflow (2006), `http://www.mathworks.com`
17. Meinicke, L., Hayes, I.: Continuous action system refinement. In: Uustalu, T. (ed.) MPC 2006. LNCS, vol. 4014, pp. 316–337. Springer, Heidelberg (2006)
18. Meyer, B.: Object-Oriented Software Construction, 2nd edn. Prentice-Hall, Englewood Cliffs (1997)
19. Mikáč, J., Caspi, P.: Temporal refinement for Lustre. In: SLAP 2005. Proceedings of Synchronous Languages, Applications and Programming, Edinburgh, Scotland. ENTCS, Elsevier, Amsterdam (2005)
20. Scilab Consortium. Scilab/Scicos (2006), `http://www.scilab.org`
21. Tiwari, A., Shankar, N., Rushby, J.: Invisible formal methods for embedded control systems. Proceedings of the IEEE 91(1), 29–39 (2003)
22. Tripakis, S., Sofronis, C., Caspi, P., Curic, A.: Translating discrete-time Simulink to Lustre. ACM Trans. on Embedded Computing Systems 4(4), 779–818 (2005)