Pontus Boström | Lionel Morel | Marina Waldén

Stepwise development of Simulink models using the refinement calculus framework

Turku Centre *for* Computer Science

TUCS Technical Report
No 821, 2007

Stepwise development of Simulink models using the refinement calculus framework

Pontus Boström
Åbo Akademi University, Department of Information Technologies
Turku Centre for Computer Science
Joukahaisenkatu 3-5, 20520 Turku, Finland
pontus.bostrom@abo.fi

Lionel Morel
project ESPRESSO
INRIA/IRISA - Campus universitaire de Beaulieu
35042 Rennes Cedex, France
lionel.morel@inria.fr

Marina Waldén
Åbo Akademi University, Department of Information Technologies
Joukahaisenkatu 3-5, 20520 Turku, Finland
marina.walden@abo.fi

TUCS Technical Report
No 821, 2007

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 systems 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.

Keywords: Simulink, Design by Contract, Refinement, Action Systems

TUCS Laboratory
Distributed Systems Design Laboratory

1 Introduction

Simulink / Stateflow [23] 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 general goal of our work (see [8, 9, 10]) is to establish such a development method by 1) proposing a model architecture [10] and 2) study 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/Stateflow models. The present paper focuses on defining this refinement for Simulink alone, while letting the extension to Stateflow for further investigation.

The refinement calculus [6] gives a good theoretical 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.

1.1 Design-by-contract for embedded controllers

Contract-based design [6, 11, 25] 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 been successfully applied to reactive programs in [20]. 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 consist 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 [4, 5] 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. Contracts together with the refinement definition also enable compositional reasoning [1] about correctness of models. Here the aim is to provide an easy to use and lightweight

reasoning framework for correctness of Simulink models.

Other formalisations of Simulink diagrams exist [3, 12, 13, 33, 34]. Each one of these take into account different subsets of Simulink. Refinement of Simulink diagrams has also been considered by Cavalcanti et al. [12]. 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 the works above. Instead of action systems, a definition of refinement [26] for Lustre [16] could also be used in conjunction with the translation from Simulink to Lustre [34]. This formalisation can only accommodate discrete models. Treating continuous models using refinement of continuous action systems [24, 29] is a rather natural extension of our formalisation. Furthermore, Simulink diagrams are similar to process networks, which has been investigated in detail before. Assume-guarantee reasoning in process networks is discussed in [27, 32]. General formal description and refinement rules for process nets using predicate transformers has also been investigated by Mahony [19]. However, we focus specifically on rules for Simulink. Control engineers usually rely on control theory [31] to analyse systems. This theory concerns mainly the dynamics of systems, such as stability, performance and robustness. Contracts in our approach concerns static properties of the system and, hence, they are used to verify different properties.

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 systems formalism and refinement calculus is, however, very versatile [6] and can accommodate these features as well.

1.2 Structure of the paper

Section 2 presents action systems as a way to describe reactive systems. Section 3 shows how we encode the Simulink block diagrams in the refinement calculus. Section 4 describes formal contract-based system design in Simulink and section 5 defines a refinement relation as well as a correctness criterion for it. Section 6 discuss system development using refinement, while Section 7 illustrates our propositions with a simple example. Finally, section 8 concludes and gives directions for further work.

2 Action Systems

Action systems [4, 5, 7] are used for describing reactive and distributed systems. The formalism was invented Back and Kurki-Sounio and inspired by Dijkstras guarded command language [14].

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 statespace of a program in the refinement calculus is assumed to be of type Σ. Predicates are func-

$\langle f \rangle.q.\sigma$	$=$	$q.f.\sigma$	(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)
$[R].q.\sigma$	$=$	$\forall \sigma' \cdot R.\sigma.\sigma' \Rightarrow q.\sigma'$	(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)

Table 1: Program statements with their predicate transformer semantics

tions from the statespace 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 statespace Σ to predicates on the input statespace 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 [6]. The functional update consists of assignments of the type $(\langle f \rangle \mathrel{\hat{=}} x := e)$, where the value of variable x in statespace 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'. Values of other variables than x in σ remains unchanged in the updated statespace σ'.

An action system consists of a set of variables, an initialisation and actions. An example of an action system $\mathcal{A}$ is shown below:

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

Here x denotes the local variables and z global variables. The initialisation action is given as a predicate A_0. All actions consists of conjunctive predicate transformers and they can be written together as one single action A 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 and infinite. Finite behaviours can be aborted or miraculous, since we do not consider action systems that can terminate normally here. In order to only consider infinite behaviours, terminated behaviours are extended with infinite sequences of $\bot$ or $\top$ depending on if the behaviour was aborted or miraculous. These states are referred to as *improper states*.

Assume that the action A can be written as $\{tA\};[nA]$, where tA is a predicate and nA is a relation that relates the old and new state-spaces. This can 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. What can be observed is a trace of a behaviour where the global variables z have been extracted to get $(z.\sigma_0, z.\sigma_1, \ldots)$. 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 remains unchanged. The semantics of action system $\mathcal{A}$ is now a set of observable traces of behaviours [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]. On the extended statespace $\Sigma \cup \{\bot, \top\}$ we define a ordering $\sigma \leq \tau$. The ordering is given 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:

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

This means that for each trace in the refined system $\mathcal{A}'$ there exists a corresponding trace in the abstract system $\mathcal{A}$.

This definition of refinement is not practical for use in proofs [5, 7]. Instead refinement can be proved using the standard notion of (forward) data refinement. Consider again two action systems $\mathcal{A}$ and $\mathcal{A}'$. Assume we have a abstraction relation R relating the statespaces of $\mathcal{A}$ and $\mathcal{A}'$. Then $\mathcal{A}$ is data refined $\mathcal{A}'$ under relation R, if the following conditions hold [5, 7]:

$$A'_0 \subseteq \{R\}.A_0$$

$$\{R\}; A \sqsubseteq A'; \{R\}$$

The first condition concerns correct refinement of the initialisation. For each possible initialisation in the concrete system, there must exist a corresponding initialisation in the abstract system. The second condition concerns refinement of the action. For each action step in the concrete system there must exist a corresponding step in the abstract system. Refinement in Lustre [26] has also been defined in a similar manner.

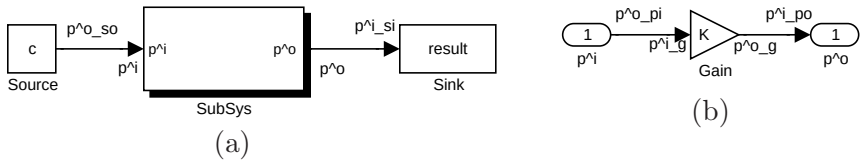


Figure 1: Example of a Simulink model. The diagram (b) shows the content of the subsystem *SubSys* in (a)

3 Encoding Simulink block diagrams as Action systems

The structure of a Simulink block diagram can be described as a set of blocks containing ports, where the 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. An example of a Simulink diagram is shown in Figure 1. The diagram contains one source block giving a value c to a signal connected to the subsystem *SubSys*. The subsystem have in- and out-blocks p^i and p^o to communicate with blocks higher in the hierarchy. The functionality of the subsystem is given by a gain block, *Gain*, that multiplies the input by a constant K. The output from the subsystem is then delivered to a sink block that consumes the given value. This diagram, hence, computes $result = Kc$.

3.1 Translating Simulink Model Elements

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

- 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. Furthermore, subsystems can be partitioned into virtual (normal) subsystems B^{vs} and non-virtual subsystems B^{ns}, $B^{ns} \cup B^{vs} = B^s$.

- $root \in B^{vs}$ is the root subsystem.

- $\mathsf{subh} : B \rightarrow B^s$ 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, $P = P^i \cup P^o$;

- $\mathsf{blk} : P \rightarrow B$ is a relation that maps every port to the block it belongs to;

- $\mathsf{sig} : P^i \nrightarrow P^o$ maps each in-port to the out-port it is connected to by a signal. Since we need to be able to analyse model fragments, all in-ports are not necessarily connected.

- $\mathsf{subi} : B^s \rightarrow (P^o \nrightarrow P^i)$ is a partial function that describes the mapping between the out-ports of the in-blocks in a subsystem and the in-ports of that subsystem.

- $\mathsf{subo} : B^s \rightarrow (P^o \nrightarrow P^i)$ is a partial function that describes the mapping between the out-ports of a subsystem and the in-ports of the out-blocks in that subsystem.

- 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. Note that Simulink does not assume that these parameters are constants and they can be modified during simulation. However, this is contrary to the dataflow philosophy of Simulink and it is not allowed here.

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

To illustrate this description, consider the Simulink block diagram given in Figure 1. The blocks are defined by $B \mathrel{\hat{=}} \{\mathit{Source},\ \mathit{SubSys},\ p^i,\ \mathit{Gain},\ p^o,\ \mathit{Sink}, \mathit{root}\}$. The subsystems are given as $B^s \mathrel{\hat{=}} \{\mathit{SubSys}, \mathit{root}\}$ and the hierarchy as $\mathsf{subh} \mathrel{\hat{=}} \{(\mathit{Gain},\ \mathit{SubSys}),\ (\mathit{SubSys},\ \mathit{root}),\ \ldots\}$. The subsystems are all virtual subsystems, $B^{vs} = B^s$. Names of ports are usually not shown in diagrams. Here we have the following ports, $P = \{p^o_{so}, p^i, p^o,\ p^i_{si}, \ldots\}$. The function describing to which block each port belongs to is then given as $\mathsf{blk} \mathrel{\hat{=}} \{(p^o_{so}, \mathit{Source}), (p^i_g,\ \mathit{Gain}),\ (p^o_g,\ \mathit{Gain}),\ (p^i,\ \mathit{SubSys}),\ \ldots\}$. The connections between the ports is defined as $\mathsf{sig} \mathrel{\hat{=}} \{(p^i,\ p^o_{so}),\ (p^i_g,\ p^o_{pi}), \ldots\}$. The relations describing how ports in in/out-blocks corresponds to ports of subsystems are given by the partial functions subi and subo. The in-port of the subsystem is related to the out-port of the in-block, $\mathsf{subi} \mathrel{\hat{=}} \{\ (\mathit{SubSys}, p^o_{pi}, \{p^i\}),\ (\mathit{SubSys}, p^o_{so}, \varnothing), \ldots, (\mathit{root}, p^o_{pi}, \varnothing),\ (\mathit{root}, p^o_{so}, \varnothing), \ldots\}$. The definition of the relation between out-ports in subsystems and in-ports in out-blocks is similar, $\mathsf{subo} \mathrel{\hat{=}} \{(\mathit{SubSys}, p^o, \{p^i_{po}\}), (\mathit{SubSys}, p^o_{so}, \varnothing), \ldots\}$. The block parameters are here $C \mathrel{\hat{=}} \{c, K\}$.

To reason about blocks, we need to express which ports depends on each other. We give a single relation for this purpose, which describes how ports

are connected to each other by signals and over subsystem boundaries.

$$\begin{aligned}\mathsf{dep} \mathrel{\hat{=}} \quad & \lambda p_1 : P \cdot \{p_2 \in P | p_1 \neq p_2 \wedge \\ & (p_1 \in (P^i) \Rightarrow p_2 = \mathsf{sig}.p_1) \wedge \\ & (p_1 \in P^o \Rightarrow (\exists b \cdot b \in B^s \wedge (\mathsf{subi}.b.p_1 = p_2 \vee \mathsf{subo}.b.p_1 = p_2)))\}\end{aligned}$$

If p_1 is an in-port then the dependency is given by the signals in the model. For subsystems the relation shows how out-ports are related to in-ports over the subsystem boundary. Consider the example in Figure 1. The relation dep would here be given as $\mathsf{dep} \mathrel{\hat{=}} \mathsf{sig} \cup \{(p^o_{pi}, p^i), (p^o, p^i_{po})\}$.

The most common structuring mechanism in Simulink is the *virtual subsystem* (from here on referred to as only subsystem). These subsystems do not affect the semantics of models and they are used purely for structuring. This means that they can be removed or introduced without changing the semantics of the models as long as the connections are preserved. We like to consider only ports in blocks that actually are significant for the behaviour of a Simulink model. The function ndep then gives the connected ports, taking into account connection over the virtual subsystem hierarchy.

$$\begin{aligned}\mathsf{ndep} \mathrel{\hat{=}} \quad & \lambda p_1 : P \cdot \{p_2 \in P^o | \\ & \mathsf{blk}.p_1 \notin (B^i \cup B^o \cup B^{vs}) \wedge \mathsf{blk}.p_2 \notin (B^i \cup B^o \cup B^{vs}) \wedge \\ & p_2 \in \mathsf{dep}^+.p_1 \wedge \\ & (\forall p \cdot p \in P \wedge p \in (\mathsf{dep}^+.p_1 \cap (\mathsf{dep}^{-1})^+.p_2) \\ & \Rightarrow \mathsf{blk}.p \in (B^i \cup B^o \cup B^{vs}))\}\end{aligned}$$

This relation states that two ports are connected, if they are in blocks significant for the behaviour of the model and there is a sequence of signals, ports in virtual subsystems, in-blocks or out-blocks connecting them. In the model in Figure 1 the relation ndep is given as $\mathsf{ndep} \mathrel{\hat{=}} \{(p^i_{si}, p^o_g), (p^i_g, p^o_{so})\}$. It relates the ports in blocks *Source*, *Gain* and *Sink*. These blocks are the only blocks needed to describe the behaviour of the model, since the subsystem is only used for structuring. This provides a way to give the semantics of a Simulink diagram in a hierarchy independent way. Note that we will not consider semantics non-virtual subsystems in this paper, but they are present in the formalisation to ensure that it can be used in future work.

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

Variables in the refinement calculus framework correspond to ports in Simulink. The function ν describes this mapping:

$$\nu : P \nrightarrow V$$

where V is a set of variable names. Only necessary ports are translated to variables, i.e., ports that are or can be in $(\mathrm{dom}\,.\mathsf{ndep} \cup \mathrm{ran}\,.\mathsf{ndep})$. The

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

Simulink construct	Requirements	Translation
Port, p:	$p \in P \wedge$ $\mathsf{blk}.p \notin (B^i \wedge B^o \wedge B^{vs})$	$\nu.p$
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:	$b \in B^{mem} \wedge$ $\mathsf{blk}.p^o = b \wedge p^o \in P^o \wedge$ $\mathsf{blk}.p^i_f = b \wedge p^i_f \in P^i \wedge$ $\mathsf{blk}.p^i_g = b \wedge p^i_g \in P^i$	$\nu.p^o := f_b.(\nu.p^i_f).x_b.c_b$, $x_b := g_b.(\nu.p^i_g).x_b.c_b$

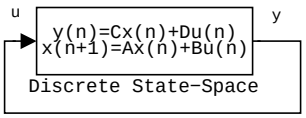
constant block parameters are translated directly to variables c. The connections between ports, $p_1 = \mathsf{ndep}.p_2$, are modelled as assignments. A normal block can contain in-ports, out-ports, and block parameters. Each block $b \in B^b$ is associated to a function f_b that updates its out-ports based on the value of the in-ports p_j 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 function g_b). These functions do not need to depend on all in-ports.

3.2 Ordering the Assignments Obtained from Simulink

Simulink has an interleaving semantics for parallel blocks. Hence, we need to find in which order blocks can be executed to give the semantics of Simulink in the refinement calculus framework. To do this, we first need to determine the dependency between all the ports in the Simulink model. We define a relation totdep to describe this.

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

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, $\mathsf{fdep} : P^o \rightarrow \mathcal{P}(P^i)$, cannot often be determined syntactically on the graphical part of the Simulink diagram. Often block parameters can affect this, e.g. the parameter D in the State-space model block in Figure 2. If $D = 0$ then the value of the out-port is not directly dependent on the in-port, $\mathsf{fdep}.y = \varnothing$. The data



(a)

$$y(k) = Cx(k) + Du(k)$$
$$x(k+1) = Ax(k) + Bu(k)$$

(b)

Figure 2: Example of a model where the data dependency is dependent of block parameter values. The diagram (a) shows the model, while (b) shows the function computed by the block

dependency for different blocks is documented in the Simulink documentation [23]. For deterministic models, 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. Simulink automatically checks that the graph is acyclic, if the option to check for algebraic loops is activated. We can now define the order in which the translated Simulink model elements can be executed.

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 Definition 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 Figure 1. The data dependency inside blocks fdep is given by $\mathsf{fdep} \mathrel{\hat{=}} \{(p_g^o, p_g^i)\}$, since the gain block is the only block where the out-port depends on the in-port. The complete ordering of ports totdep can then be given as $\mathsf{totdep} \mathrel{\hat{=}} \{(p_{si}^i, p_g^o), (p_g^o, p_g^i), (p_g^i, p_{so}^o)\}$. The refinement calculus representation of M is then denoted by $\mathsf{refCalc}.M$:

$$\begin{aligned} \mathsf{refCalc}.M \mathrel{\hat{=}} \quad & \nu.p_{so}^o := c; \\ & \nu.p_g^i := \ \nu.p_{so}^o; \nu.p_g^o := K(\nu.p_g^i); \\ & \nu.p_{si}^i := \nu.p_g^o \end{aligned}$$

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. However, this simple diagram does not have memory.

4 Specification of Simulink models

When developing Simulink models, we would like to start with an abstract overview of the system that is then refined in a stepwise manner. We use contracts to give an abstract description of the system. A contract consists of a pre-condition and a post-condition that state properties about a part

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

Table 3: Refinement calculus semantics of contract conditions.

of a Simulink model. The structure of the model can be outlined by giving the main subsystems in the model as abstract specifications described by contracts. Abstract specifications are then refined individually to model fragments satisfying their contracts.

4.1 Specification of Block Parameters

The blocks in a model usually use parameters from the Matlab workspace. These parameters are often required to have certain properties. To describe these properties we give a predicate describing the valid parameter values Q^{param}. In the refinement calculus the parameters are considered normal local variables that are only assigned in the initialisation. The valid parameter assignments are modelled as a initialisation predicate $Q^{param}(c)$ in the action systems formalism as shown in Table 3. The parameter assignments in the implementation should refine this condition.

4.2 Specification of models

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 be refined to a desired implementation. A specification block, M_s, contains in-ports p^i, out-ports p^o, pre-condition Q^{pre} and 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.

Since the ordering rules for statements in Definition 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, $S; \{Q(p_1, \ldots, p_n)\}$. □

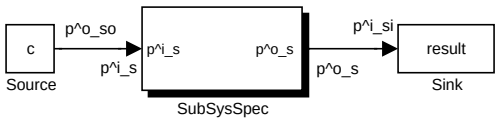


Figure 3: Example of a specification of a subsystem in Simulink. The diagram has one specification block *SubSysSpec*.

Consider the specification *SubSysSpec* in Figure 3. The specification block is here given by a subsystem with only the desired in-ports p_s^i and out-ports p_s^o. The subsystem is identified as a specification in order to ensure that its ports are translated to the refinement calculus representation. It would also be possible to create specialised blocks for the purpose of specification. Currently, tool support is being developed for a convenient and easy to use implementation of contracts. A contract for the specification block in Figure 3 is for example

$$Q^{pre} \mathrel{\hat{=}} p_s^i \geq 0$$
$$Q^{post} \mathrel{\hat{=}} p_s^o \geq p_s^i$$

This model is not executable, but it can be analysed in other ways. The refinement calculus translation of the model is then:

$$\nu.p_{so}^o := c;$$
$$\nu.p_s^i := \ \nu.p_{so}^o; \{p_s^i \geq 0\}; [\nu.p_s^o := v | v \geq p_s^i];$$
$$\nu.p_{si}^i := \nu.p_s^o$$

The specifcation *SubSysSpec* is translated to an assert statement and a non-deterministic update according to the contract. Other parts of the model are translated as in Figure 1.

In order for specification blocks to be versatile enough, the connections involving these blocks should be allowed to form cycles, i.e., use feedback. Consider the Simulink model in Figure 4 consisting of specification block M_s with pre-condition Q^{pre} and post-condition Q^{post}. A cycle can be treated as the fix-point [19], where $p^o = p^i$. This is modelled as the program $S \mathrel{\hat{=}} [p^i := v | true]; [\nu.p^o := v | Q^{post}]; [p^i = p^o]; \{Q^{pre}\}$. The idea is to force the input and output to have the same value, by adding this constraint as an assumption. Note that we need to show that the this program is feasible to avoid miraculous behaviour.

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

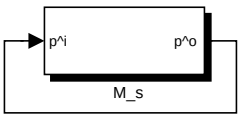


Figure 4: Example of a specification block M_s with feedback.

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, $[\nu.p_s^i = \nu.p^o]; \{Q^{pre}\}$. □

This enables us to prove certain properties about cyclic specifications. In particular it will be useful for refinement. Cycles are not allowed in the implementation, due to unpredictability of computational time and the built in non-determinism of the results [19]. Therefore, the final implementation must be shown to be without cycles. When the model is acyclic the assumption $[p^i = p^o]$ will become trivially true and can be removed, since $[true] = skip$.

4.3 Models of physical systems

Model-based design is a popular method for designing control systems using Simulink. This means that the physical system to be controlled is also modelled. The performance of the controller can then be evaluated by simulating the complete system using the simulation tools in Simulink. The model of the physical system or environment is usually continuous and described using a system of differential equations. However, continuous or multi-rate models are not considered here. We assume that the controller samples the sensor values and we only consider what can be observed from the controller. The controller reads the sensors $p_1^s, \ldots, p_n^s$, where each sensor is an in-port, $p_j^s \in P^i$. Actuators $p_1^a, \ldots, p_m^a$ are used by the controller to direct the plant. Each actuator p_k^a is a out-port, $p_k^a \in P^o$. The specification of the plant can then again be given as pre-condition $Q^{pre}(p_1^a, \ldots, p_m^a, c)$ on the actuators and a post-condition $Q^{post}(p_1^s, \ldots, p_n^s, p_1^a, \ldots, p_m^a, c)$ on the sensors. This models what can be observed of the environment in the controller. The conformance of the plant model to the specification can be validated by simulation. However, care should be taken not to make too many assumptions about the behaviour of the environment, since its behaviour is outside the control of the software implementation. Creation of formal specifications for control systems is discussed in detail by Hayes et al. [18].

4.4 Action System Semantics of Simulink Models

Above we have given the semantics of all the needed parts of Simulink in the refinement calculus framework. The behaviour of the complete diagram is now given as an action system.

The ordering of statements is not unique and the order is significant. 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 Definitions 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. $\exists \sigma \cdot (S_k; \ldots; S_l).false.\sigma \wedge \neg(S_r; \ldots; S_s).true.\sigma$
2. $\exists \sigma \cdot \neg(S_k; \ldots; S_l).true.\sigma \wedge (S_r; \ldots; S_s).false.\sigma$
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 sequences of statements (case 3).

Since a model should be non-terminating we can consider both miraculous and aborting behaviour as erroneous behaviour that should be avoided. Hence, we need to consider only one ordering. The action system is then:

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

The global variables giving the observable behaviour are given by the ports, $p_1, \ldots, p_n$ of the initial specification. 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 $S_k; \ldots; S_l$ updating ports, which satisfy the ordering rules in Definitions 1-3. This sequence is followed by statements $R_1; \ldots; R_m$ updating the memory $x_1, \ldots, x_m$. 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.

$$\forall t \cdot t \in tr(\mathcal{M}^V) \Rightarrow t \notin \{\bot, \top\}$$

4.5 Correctness of Simulink models

The aim of this paper is to define and to show how to 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 here 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 $(S_1, \ldots, S_n, \{Q^{pre}\}, \{Q^{post}\}, R_1, \ldots, R_m)$. These statements are then ordered according to the rules in Definitions 1-3. This is illustrated by the possible refinement calculus translation $\mathsf{refCalc}.M$ of M below:

$$\mathsf{refCalc}.M = \; S_k; \ldots; S_j; \ldots; \{Q^{post}\}; \ldots; S_l; R_1; \ldots; R_m$$

The assert statement formed from the pre-condition Q^{pre} cannot be included, since not all in-ports are connected. Model M is therefore a *partial model*. However, below we show how to verify the model 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: $(S_1, \ldots, S_n, \{Q^{pre}\}, \{Q^{post}\}, [\nu.p_f^i := v|Q^{pre}], R_1, \ldots, R_m)$. The validation model for M is given as:

$$\begin{array}{l}\mathsf{refCalc}.M^V \mathrel{\hat{=}} \\ \quad S_k; \ldots; [\nu.p_f^i := v|Q^{pre}]; \ldots; \{Q^{pre}\}; \ldots; S_j; \ldots; \{Q^{post}\}; \ldots; S_l; \\ \quad R_1; \ldots; R_m\end{array}$$

□

Note that since some ports of the pre-condition is already connected, parts of the model is executed before the pre-condition. 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 aborted or miraculous traces. The correctness of the validation model can be checked using model checking, other verification tools or testing. A test case is a model where the statement $[\nu.p_f^i := v|Q^{pre}(v, \nu.p_b^i)]$ 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 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_j^{pre} and post-condition Q_j^{post}, the correctness constraints can be simplified. There is no need to iterate the system over time, since the model does not contain memory blocks and the execution of the graph is independent of the number of times it has been executed before (see Definitions 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 model fragments to prove that the connections between them are correct. We need to verify that:

1. The assert conditions are not violated, $M^V.true = true$.
2. The validation model does not behave miraculously, $M^V.false = false$.

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

$$([Q^{param}(c)]; [\nu.p_f^i := v | Q^{pre}]; \{Q_1^{pre}\}; [\nu.p_1^o := v | Q_1^{post}]; \ldots; \\ \{Q_m^{pre}\}; [\nu.p_m^o := v | Q_m^{post}]; \{Q^{post}\}).true$$

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

5 Refinement

We would like to develop systems using Simulink and stepwise refinement. In order to get a definition of refinement, we use the translation of Simulink to the Action Systems formalism. The most important type of refinement is the refinement of abstract specifications given as pre- and post-conditions. However, we can also use superposition refinement to add features, while preserving old features. Since virtual subsystems do not have any semantics, refactoring of the model hierarchy can also be considered to be a refinement. The properties of the block parameters of the model was given using an initialisation condition. Block parameters can be added during the refinement, since parameters are treated as local variables only modified in the initialisation. The initialisation condition can also be strengthened.

5.1 Refinement of Block Parameters

We start with the description of refinement of block parameters. New parameters can be added in the refined model M' and the initialisation of the parameters can be refined. Let b be the new block parameters in the refined model M' and $Q^{param\prime}(a', b')$ be the initialisation predicate for the parameters in the new model. For the initialisation to be valid, we need to ensure that the assignment is still feasible $\exists a, b \cdot Q^{param\prime}(a, b)$ and that the new assignment refines the old. The rule for refinement of initialisation is given in Subsection 2.2. Here the abstraction relation is given as $R \mathrel{\hat{=}} (a' = a)$.

$$Q^{param\prime}(a', b') \Rightarrow \exists a \cdot a' = a \wedge Q^{param}(a)$$

If the condition is of the form $Q^{param\prime}(a', b') \mathrel{\hat{=}} Q^{param}(a') \wedge Q_b^{param}(a', b')$, the condition is a refinement by construction. When the parameters are given concrete values in the implementation, it is sufficient to check that these values satisfy the initialisation condition.

5.2 Refinement of Abstract Specifications

An abstract specification is given as a block containing only ports associated with a pre-condition and a post-condition. In the refinement the abstract specification is replaced with a new model fragment. This model fragment

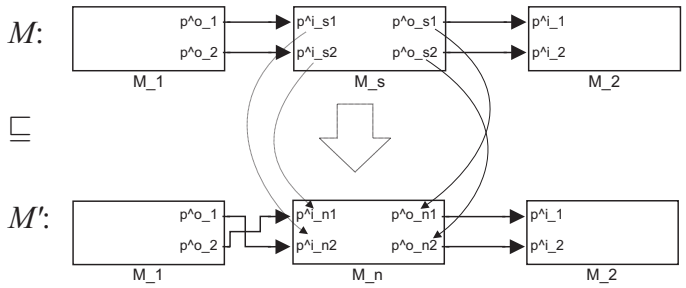


Figure 5: Illustration of refinement of abstract specification M into refinement M'.

should then be shown to be a refinement of the specification. First we need to determine how blocks and ports in the abstract diagram M relates to the to ports in the refined diagram M'. Two blocks are equal in the two models, if they have the same name and type. Two ports are equal, if their parent blocks are equal and they have the same names.

5.2.1 Construction of the refined model.

Consider a specification block M_s with in-ports P_s^i, out-ports P_s^o, pre-condition Q^{pre} and post-condition Q^{post} in a model $M = (B, root, \mathsf{subh}, P, \mathsf{blk}, \mathsf{sig}, \mathsf{subi}, \mathsf{subo}, C)$. This specification is refined by the model fragment $M_n = (B_n, root_n, \mathsf{subh}_n, P_n, \mathsf{blk}_n, \mathsf{sig}_n, \mathsf{subi}_n, \mathsf{subo}_n, C_n)$ with pre-condition Q_n^{pre}. The refinement is illustrated in Figure 5. 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. The new Simulink model M' that we obtain is given as follows:

- $B' = (B - (|\mathsf{subh}|^{-1})^*.M_s) \cup (B_n - \{root_n\})$ is the blocks in the new model. B_n gives the new blocks that replace blocks from M_s.

- $P' \subseteq P \cup P_n$. The ports from both models are preserved except for the ports in the subsystem that is refined. If a port is in a block that is present in both M and M', the port is also present in both models, $\forall p \cdot p \in P \wedge \mathsf{blk}.p \in B \cap B' \Rightarrow p \in P'$. On the other hand, if the port is in a block in M that is no longer present in M', the port is not present in M' either, $\forall p \cdot p \in P \wedge \mathsf{blk}.p \notin (B - B') \Rightarrow p \notin P'$. The ports from the new model M_n are all included in M', $\forall p \cdot p \in P_n \Rightarrow p \in P'$.

- $\mathsf{sig}' : P^{i\prime} \rightarrow P^{o\prime}$. The new parent relation sig' is the same as sig except for connections that involves ports that are not included in M', $p \in (P - P')$ (see Figure 5),

$$\forall p \cdot p \in (P^{i\prime} \cap P^i) \wedge \mathsf{sig}.p \in (P^{o\prime} \cap P^o) \Rightarrow \mathsf{sig}'.p = \mathsf{sig}.p$$

Whenever sig_n is defined in M_n it should be preserved,

$$\forall p \cdot p \in P_n^i \wedge p \in \mathrm{dom}\,.\mathsf{sig}_n \Rightarrow \mathsf{sig}'.p = \mathsf{sig}_n.p$$

Connections involving the specification M_s are replaced by connections involving the new ports from M_n (see Figure 5). The first requirement on the connections is that they are not allowed to introduce more unconnected ports, $\forall p \cdot p \in P^i \wedge p \in \mathrm{dom}\,.\mathsf{sig} \Rightarrow p \in \mathrm{dom}\,.\mathsf{sig}'$. We also have that every in-port connected to an out-port of the abstract specification M_s in M is connected to an out-port from the new model M_n in M'. There should be as many out-ports connected from M_n to the rest of M' as there were out-ports in M_s connected to M. This is needed to obtain a refinement relation.

$$\forall p^i \cdot p^i \in P^i \wedge \mathsf{sig}.p^i \in P_s^o \Rightarrow \exists p^o \cdot p^o \in P_n^o \wedge p^o = \mathsf{sig}'.p^i \ ,$$

$$\mathrm{card}\,.P_s^o = \mathrm{card}\,.\{p \in P_n^o | \exists p^i \cdot p^i \in P^i \wedge p = \mathsf{sig}'.p^i\}$$

Each in-port from the specification $p \in P_s^i$ has to have a corresponding port from P_n^i (see Figure 5). Hence, we have that every out-port connected to M_s is connected to a port in M_n in the refined model.

$$\begin{array}{l} \forall p^i \cdot p^i \in P^i \wedge p^i \in P_s^i \\ \quad \Rightarrow \exists p_n^i \cdot p_n^i \in P_n^i \wedge p_n^i \notin \mathrm{dom}\,.\mathsf{sig}_n \wedge \mathsf{sig}.p^i = \mathsf{sig}'.p_n^i \end{array}$$

- $\mathsf{blk}' : P' \rightarrow B'$. The ports in the new model M' are either in the unchanged blocks as given by M or in the blocks given by M_n. We have the following restrictions on blk':

$$\forall p \cdot p \in (P' \cap P) \Rightarrow \mathsf{blk}'.p = \mathsf{blk}.p \ ,$$

$$\forall p \cdot p \in P_n \Rightarrow \mathsf{blk}'.p = \mathsf{blk}_n.p$$

- $\mathsf{subh}' : B' \rightarrow B^{s\prime}$. We have that the hierarchy is preserved and that the specification M_s is replaced by the model fragment M_n.

$$\begin{array}{l} \forall b \cdot b \in B - \{M_s\} \Rightarrow \mathsf{subh}'.b = \mathsf{subh}.b \ , \\ \forall b \cdot b \in (B_n - \{root_n\}) \Rightarrow \mathsf{subh}'.b = \mathsf{subh}_n.b \ , \\ \forall b \cdot b \in B_n \wedge b \neq root_n \wedge \mathsf{subh}_n.b = root_n \Rightarrow \mathsf{subh}'.b = \mathsf{subh}.M_s \end{array}$$

  The specification M_s is replaced by new blocks that are at the top level in M_n, $\mathsf{subh}_n.b = root_n$.

- subi and subo have the same restrictions as blk. The relation fdep describing how out-ports depends on in-ports inside basic blocks also has the same restrictions as blk.

- $C' = C \cup C_n$, where initialisation condition for the block parameters becomes $Q^{param\prime} = Q^{param} \wedge Q_n^{param}$.

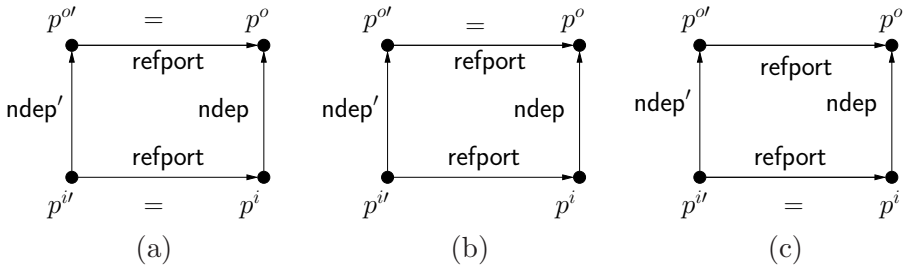


Figure 6: Relations between ports in the refined and abstract diagrams.

5.2.2 Verifying the correctness of the refined model.

Above we described how the refinement M' is constructed from the abstract model M and model fragment M_n. The refined model M' is then mapped to constructs in the refinement calculus as previously to give the refinement definition. The refinement M' contains more ports and therefore more variables than M. However, global variables cannot be added in the refinement and new ports are instead translated to local variables. In the refinement we now have two functions ν' and ν'_{loc} that maps ports to variables. The function ν' maps ports corresponding to ports in the initial specification to global variables and the function ν'_{loc} maps new ports to local variables.

First we derive a refinement relation $\mathsf{refport} : P' \nrightarrow P$ between ports. This is a partial bijective function, which has the following restrictions:

$$
\begin{aligned}
&\forall p' \cdot p' \in P' \land p' \in P \land \mathsf{ndep}'.p' \in P \Rightarrow p' = \mathsf{refport}.p' \ , \\
&\forall p' \cdot p' \in P^{i\prime} \land p' \notin P \land \mathsf{ndep}'.p' \in P \\
&\quad \Rightarrow \exists p \cdot p \in P \land \mathsf{ndep}'.p' = \mathsf{ndep}.p \land p = \mathsf{refport}.p' \ , \\
&\forall p' \cdot p' \in P^{i\prime} \land p' \in P \land \mathsf{ndep}.p' \notin P \Rightarrow \mathsf{refport}.\mathsf{ndep}'.p' = \mathsf{ndep}.p'
\end{aligned}
$$

The first case handles preservation of the graph in the refinement. This is illustrated in figure 6 (a). The second case concerns the refinement of the in-ports of the abstract specification block in the diagram M. Here two ports corresponds to each other if they have the same predecessor, $\mathsf{ndep}'.p' = \mathsf{ndep}.p$ (see figure 6 (b)). Note that this relation is not unique, but the requirement that refport should be bijective ensures that there is a one-to-one mapping. The last case concerns refinement of the out-ports of the abstract specification block. Two ports are the same if they have a common successor, $\mathsf{refport}.\mathsf{ndep}'.p' = \mathsf{ndep}.p'$ (see figure 6 (c)).

Each port in M that is translated to a variable has a corresponding port in M'. Either the ports are the same in both models or if the port belonged to block M_s, ports from M_n are used instead. The mapping of variables ν' in the refinement is defined as follows:

$$\forall p' \cdot p' \in P' \land p' \in \mathrm{dom}\,.\mathsf{refport} \Rightarrow \nu'.p' = \nu.\mathsf{refport}.p'$$

Ports related via refport maps to the same global variables, as shown in figure 7

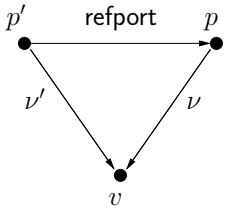


Figure 7: The figure in shows how ports are mapped to variables.

Ports in the refinement that cannot be mapped to global variables from the abstract specification are mapped to local variables. This mapping is denoted by $\nu'_{loc} : P' \nrightarrow V'$. The domain of this function is given as, $(\mathsf{dom}.\nu'_{loc} = \{p \in P' | \mathsf{blk}.p \notin (B^{i\prime} \cup B^{o\prime} \cup B^{vs'}\} - \mathsf{dom}.\mathsf{refport})$, since ν'_{loc} maps all remaining ports in M' to variables.

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. To verify that the refinement is correct, we check it in an environment where Q^{pre} holds. This context information [6] can be used, since other components are guaranteed to maintain it. First we note that we can add an assert statement $\{Q^{post}\}$ after the statement $[\nu.p^o_{s1}, \ldots, \nu.p^o_{sn} := v | Q^{post}]$ in the abstract specification, without changing its behaviour. In the refinement, the contract statements $(\{Q^{pre}\}, [\nu.p^o_{s1}, \ldots, \nu.p^o_{sn} := v | Q^{post}])$ 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 $([\nu.p^i_{s1}, \ldots, \nu.p^i_{sm} := v | Q^{pre}], \{Q^{post}\}, \{Q_n^{pre}\}, S_1, \ldots, S_m, R_1, \ldots, R_t)$. This model, $\mathsf{refCalc}.M_n^V$, is constructed from statements above ordered according to the rules in Definitions 1-3. Note that statement $[\nu'.p^i_{s1}, \ldots, \nu'.p^i_{sm} := v | Q^{pre}]$ 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.

$$\begin{aligned} \mathsf{refCalc}.M_n^V \;\hat{=}\; & S_k; [\nu'.p^i_{s1}, \ldots, \nu'.p^i_{sm} := v | Q^{pre}]; \ldots; \\ & \{Q_n^{pre}\}; \ldots; S_l; \{Q^{post}\}; \\ & R_1; \ldots; R_t \end{aligned}$$

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 $[\nu.p^o_{s1}, \ldots, \nu.p^o_{sn} := v | Q^{post}] \sqsubseteq \mathsf{refCalc}.M_n$.

- 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, $[\nu'.p^i_{s1}, \ldots, \nu'.p^i_{sm} := v | Q^{pre}]$.
- If $[\nu.p^o_{s1}, \ldots, \nu.p^o_{sn} := v | Q^{post}] \sqsubseteq \mathsf{refCalc}.M_n$ 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 $\mathcal{M}_n^V$ does not abort we can conclude that $\{Q^{pre}\} \sqsubseteq \{Q_n^{pre}\}$ and $[\nu.p^o_{s1}, \ldots, \nu.p^o_{sn} := v | Q^{post}] \sqsubseteq \mathsf{refCalc}.M_n$ must hold. ■

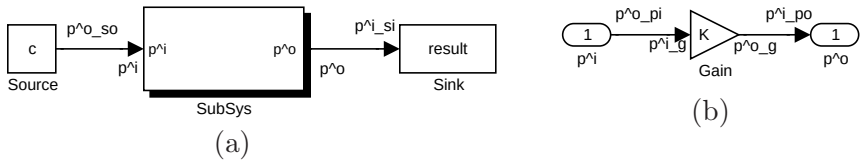


Figure 8: Example of a Simulink model that refines the model in Figure 3. The model is identical to the one in Figure 1.

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

As an example, consider refinement of the Simulink model in Figure 3 with an abstract specification *SubSysSpec*. One possible refinement of this model is the model in Figure 8. There the specification *SubSysSpec* is replaced by the implementation *Subsys*. The function mapping old ports to new ports is given as $\mathsf{refport} = \{(p^o_{so}, p^o_{so}), (p^i_g, p^i_s), (p^o_g, p^o_s), (p^i_{si}, p^i_{si})\}$. The ports are mapped to variables using the functions ν' and ν_{loc}, which are partial functions with the following domains; $\mathrm{dom}\,.\nu' \mathrel{\hat{=}} \{p^o_{so}, p^i_g, p^o_g, p^i_{si}\}$ and $\mathrm{dom}\,.\nu_{loc} \mathrel{\hat{=}} \varnothing$. This means that in this case there are no local variables obtained from the ports.

5.3 Superposition refinement

Superposition refinement is a convenient refinement method where new features are added to the model, while old features remain unchanged. We can derive superposition rules involving only syntactic constraints for Simulink models. The main idea is to preserve all the behaviour of the old model, while adding new blocks and connection between them. Consider again a model M where features M_n are added to obtain a refined model M'. We have the following requirements on M_n.

- Unconnected ports of M_n should have no pre-condition.
- M_n should be correct, i.e., the validation model $\mathcal{M}^V_n$ should not have any improper traces.

The refinement model M' can then be constructed as follows.

- Basic blocks, merge blocks, memory blocks and non-virtual subsystems from M are preserved and new blocks from M_n are only added in the refinement M', $(B^{b\prime} = B^b \cup B^b_n) \wedge (B^{m\prime} = B^m \cup B^m_n) \wedge (B^{mem\prime} = B^{mem} \cup B^{mem}_n) \wedge (B^{ns\prime} = B^{ns} \cup B^{ns}_n)$. New virtual subsystems B^{vs}, in-blocks B^i and out-blocks B^o can be introduced and removed.
- P' and blk'. The ports in the blocks from the old model M and the new model fragment M_n remains in the same blocks, except for ports in subsystems, in-blocks and out-blocks.

$$\forall p \cdot p \in P \wedge \mathsf{blk}.p \notin (B^i \cup B^o \cup B^{vs}) \Rightarrow \mathsf{blk}'.p = \mathsf{blk}.p\ ,$$
$$\forall p \cdot p \in P_n \wedge \mathsf{blk}_n.p \notin (B^i_n \cup B^o_n \cup B^{vs}_n) \Rightarrow \mathsf{blk}'.p = \mathsf{blk}_n.p$$

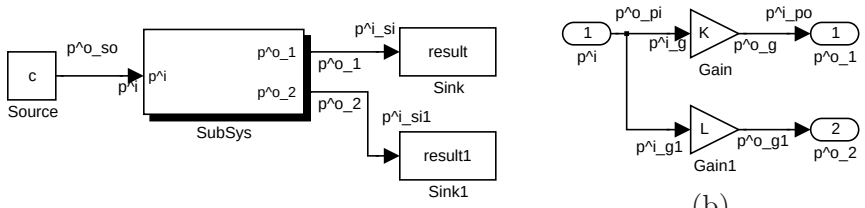


Figure 9: Example of superposition refinement of the model from Figure 1. Again, the content of the subsystem in (a) is shown in (b)

- sig', subh', subi' and subo' are chosen in such a manner that the graph formed by the ports in the old model and the new model are preserved. Furthermore, there can be no data flowing from ports in M_n to ports in M. This means that if a port p from M is connected in M', then it was already connected in M. This gives the following conditions for ndep:

$$\begin{aligned}
&\forall p \cdot p \in P \wedge p \in \mathrm{dom}\,.\mathsf{ndep} \Rightarrow \mathsf{ndep}'.p = \mathsf{ndep}.p\ ,\\
&\forall p \cdot p \in P_n \wedge p \in \mathrm{dom}\,.\mathsf{ndep}_n \Rightarrow \mathsf{ndep}'_n.p = \mathsf{ndep}.p\ ,\\
&\forall p \cdot p \in P \cap P' \wedge p \in \mathrm{dom}\,.\mathsf{ndep}' \Rightarrow p \in \mathrm{dom}\,.\mathsf{ndep}
\end{aligned}$$

- $\nu' = \nu$, since all old ports are still present
- $C' = C \cup C_n$, where initialisation condition for the block parameters becomes $Q^{param\prime} = Q^{param} \wedge Q_n^{param}$.

Theorem 2 (Correctness of superposition refinement) *The model M' is a correct refinement of M, $M \sqsubseteq M'$* □

Proof Since the new parts from M_n do not abort and have no pre-conditions, the new parts from M_n will not contribute with any aborted traces. The behaviour of the old model M is preserved in M', since all non-virtual blocks are preserved and ndep is also preserved. If M is partial, there are no connections from the new parts M_n to the old parts M in M'. This means that the new parts do not affect the old behaviour. ■

Note that the refinement constraints only concern the graph formed by the Simulink model. Correctness of superposition refinement can, hence, be verified by checking only syntactic constraints.

An example of superposition refinement of the Simulink model in Figure 1 is given in Figure 9. A new gain block $Gain1$ and a new sink block $Sink1$ are added. To connect the new blocks with the old model a new port need to be added to the subsystem $SubSys$. We have that the mapping from ports to global variables is preserved, $\nu' = \nu$, and that all new ports are mapped to local variables, $\mathrm{dom}\,.\nu'_{loc} = \{p^i_{g1}, p^o_{g1}, p^i_{si1}\}$.

5.4 Subsystem introduction and removal

We first note that as long as the refinement calculus translation of the model remains the same, the model can be rewritten. We can thus introduce or remove subsystem blocks in M to obtain a model M' as long as the following properties holds.

- Basic blocks, merge blocks, memory blocks and non-virtual subsystems from M are preserved in M', $(B^{b\prime} = B^b) \wedge (B^{m\prime} = B^m) \wedge (B^{mem\prime} = B^{mem}) \wedge (B^{ns\prime} = B^{ns})$. Only virtual subsystems B^{vs}, in-blocks B^i and out-blocks B^o can be added or removed.
- P'. Every port not in a subsystem, in-block or out-block remains in the same block after the refactoring,

$$\forall p \cdot p \in P \wedge \mathsf{blk}.p \notin (B^i \cup B^o \cup B^{vs}) \Rightarrow \mathsf{blk}.p = \mathsf{blk}'.p$$

- $\mathsf{ndep}' = \mathsf{ndep}$, the connections between ports that are translated to variables in the refinement calculus are preserved.

Theorem 3 (Correctness of refactoring) *The model M and M' are equivalent, $M \sqsubseteq M'$ and $M' \sqsubseteq M$.* □

Proof All non-virtual blocks are preserved and the connections between them are also preserved. Hence, the refinement calculus translation of the two models are functionally equivalent. ■

This means that the same ports are connected in the new model, but via possibly different in- and out-ports of virtual subsystems. This type of refinement can also be checked by examining the graph formed by the Simulink models. This is a special form of superposition refinement with empty M_n.

6 Development process

Above we have given a definition of refinement and a way to create contracts for Simulink models. To fully use these features a good development process is needed. The idea of refinement is to first create an abstract specification using contracts that captures the most essential properties of the system. This specification is then refined to eventually obtain an implementation that can be executed.

Contracts in the form given in this paper are not without problems. This framework can only express safety properties easily. However, control performance and real-time properties are often important in control applications. These properties cannot usually be analysed using the contracts described here. This can lead to underspecified systems, where some properties are not captured by the specification. On the other hand, verification of properties that are stated is significantly simplified due to the simple form of the

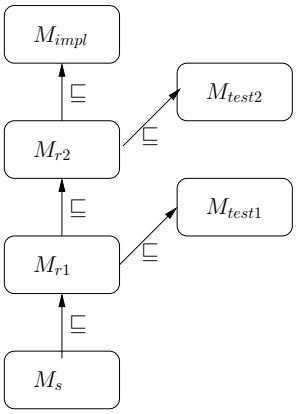


Figure 10: Overview of the development process for developing applications using refinement

contracts. We have to take into account these limitations when developing systems using refinement. We propose the following method, illustrated in Figure 10:

1. Start with an abstract specification (M_s) outlining the most important parts of the system. Contracts are used to abstractly describe the behaviour and responsibilities of the different parts. Both the controller and its environment should be taken into account.

2. The system is refined in a stepwise manner. Specifications are decomposed into more detailed models using refinement of specifications. It is also desirable to check that partially finished models (M_{r1} and M_{r2}) conforms to requirements not expressed in the contracts, e.g. performance requirements. To do this, it is possible to create *test models* (M_{test1} and M_{test2}) that can be simulated (executed) and, hence, used to investigate such properties. These models are refinements of the system developed so far, but they are not necessarily refined further. They are only used to get an idea of how the properties not expressed by contracts will work in the final complete system.

3. Step 2 is repeated until all specifications have an implementation (M_{impl}). The system should then satisfy all the properties stated in the contracts.

A good way to implement contracts is to use empty subsystems as placeholders. These subsystems are associated with a contract and represents the abstract specification. During the refinement process content is added to the subsystems. Each level in the subsystem hierarchy, hence becomes a new refinement step.

7 Example of System Development Using Refinement

To illustrate the development process for construction of Simulink models using refinement, we here develop a simplified version of the classical Steam boiler system [2]. The system in the example consists of a boiler for producing steam. Water is delivered to the system using a pump that can either be switched on or off. Steam is taken from the boiler using a valve that is either fully opened or fully closed. The objective of the controller is to keep the water level between the lower limit L_1 and upper limit L_2. The controller can read the current water level using the sensor *w_level* and it can control if the pump and the out-valve of the boiler. This is modelled by the actuators *pump_on* and *out_open*, respectively. The following safety requirements are given for the water level in the controller:

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

An overview of the complete system involving both controller and plant is given in Figure 11 (a). This model consist 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 level L_1 and L_2. The parameter condition is given by:

$$Q^{param} \mathrel{\hat{=}} L_1 > 0 \wedge L_2 > L_1$$

Water levels are positive and the upper level L_2 is higher than the lower level L_1. The contract of the controller is given from the safety requirements above.

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

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

The complete diagram in Figure 11 (a) form a cycle of specifications. The following program in the refinement calculus is then a possible translation:

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

The pump and valve status ($pump\_on'$ and out_open') that the plant uses need to be the same as the ones computed by the controller, *pump_on* and *out_open*. To illustrate how a system can be refined, consider the refinement of the plant into deterministic difference equation that updates w_level' according to $w\_level' = f.x$ and the internal memory x according to $x' =$

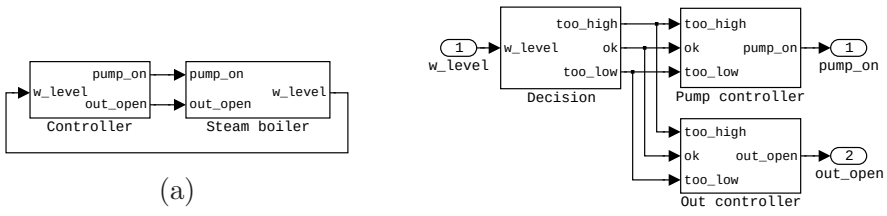


Figure 11: Simulink diagrams for the steam boiler example. An overview of the complete steam boiler system is shown in (a), while the refinement of the controller is shown in (b).

$g.x.pump\_on.out\_open$. The program obtained for the complete diagram is now:

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

Note that we usually do not write out assignments corresponding to signals (e.g. $w\_level' := w\_level$ above) and usually w_level' is substituted with w_level directly. It is easy to see that if the abstract plant specification $[w\_level' := v | Q_p^{post}]$ is correctly refined, then $System$ will be refined by $System'$. The assumption $[pump\_on' = pump\_on \wedge out\_open' = out\_open]$ can now also be removed, since it is trivially true. Note that the cycle cannot be broken in a arbitrary place, if the intention is to refine the diagram into a acyclic implementation. The cycle need to be broken before the specification that introduces the acyclic behaviour, as done in the example.

The controller is refined in a stepwise manner to obtain an implementation. Here we do the development in one single model. 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 Figure 11 (b). The first subsystem, *Decision*, decides if the water level is too high (too_high), suitable (ok) or too low (too_low). The second subsystem, *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.

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

The contract for the specification block *Pump Controller* states that the block assumes 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 the water level is too high.

$$\begin{array}{l} Q^{pre}_{pump} \mathrel{\hat{=}} too\_high \vee ok \vee too\_low \\ Q^{post}_{pump} \mathrel{\hat{=}} (too\_high \Rightarrow \neg pump\_on) \wedge \\ \qquad\qquad (too\_low \Rightarrow pump\_on) \end{array}$$

The contract for the last specification, *Out Controller*, is defined similarly:

$$\begin{array}{l} Q^{pre}_{out} \mathrel{\hat{=}} too\_high \vee ok \vee too\_low \\ Q^{post}_{out} \mathrel{\hat{=}} too\_high \Rightarrow out\_open \\ \qquad\qquad too\_low \Rightarrow \neg out\_open \end{array}$$

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 Figure 11 (b) refines the specification *Controller* we create a validation model as in Subsection 4.5.

$$\begin{array}{l} \mathsf{refCalc}.Controller \mathrel{\hat{=}} [too\_high, ok, too\_low := v_h, v_o, v_l | Q^{post}_d]; \\ \qquad\qquad \{Q^{pre}_{pump}\}; [pump\_on := v | Q^{post}_{pump}]; \\ \qquad\qquad \{Q^{pre}_{out}\}; [out\_open := v | Q^{post}_{out}]; \end{array}$$

The definition of refinement then gives the validation model that need to be verified.

$$\begin{array}{l} \mathsf{refCalc}.Controller^V \mathrel{\hat{=}} \\ \quad [w\_level := v | Q^{pre}_c(v)]; \{Q^{pre}_d\}; \mathsf{refCalc}.Controller; \{Q^{post}_c\} \end{array}$$

We first need to show that the validation model does not behave miraculously, $(\mathsf{refCalc}.Controller^V).false = false$. 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:

$$\begin{array}{l} Q^{param} \wedge Q^{pre}_c \Rightarrow Q^{pre}_d \\ Q^{param} \wedge Q^{pre}_c \wedge Q^{post}_d \Rightarrow Q^{pre}_{pump} \\ Q^{param} \wedge Q^{pre}_c \wedge Q^{post}_d \Rightarrow Q^{pre}_{out} \\ Q^{param} \wedge Q^{pre}_c \wedge Q^{post}_d \wedge Q^{post}_{pump} \wedge Q^{post}_{out} \Rightarrow Q^{post}_c \end{array}$$

This refinement of the specification *Controller* is still abstract and not executable. To illustrate the final implementation consider the implementation of the specification *Pump Controller* in Figure 12. The requirements did not state what the behaviour of the pump should be when the water level w_level is between L_1 and L_2. Here we have taken the approach to only switch on or off the pump when a water level limit is reached. Other

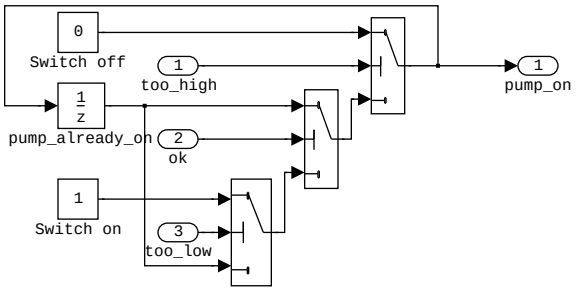


Figure 12: Implementation of the pump controller in the steam boiler system

approaches such as e.g. keeping the level as high or low as possible are also possible. 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. Note that the pump controller only uses booleans and is now suitable to be verified using model checkers.

8 Conclusions and Future Work

In this paper we have given a definition of refinement of Simulink diagrams using an action systems semantics. First we defined a translation from Simulink to action systems. Then we provided a definition of contracts for Simulink model fragments using pre- and post-conditions. The action systems formalism provided semantics to these contracts. Using the action systems semantics, rules for composing Simulink models based on the contracts were given. We then showed how an abstract specification given as a contract could be refined into an implementation satisfying the contract. Validation of the refinement could be performed by model checking or testing a *validation model*. We also provided superposition refinement rules and refactoring rules that could be verified using only syntactic constraints. Finally we discussed a development process and provided a small example to illustrate how a system can be developed using refinement.

All features of Simulink are not supported here. As future work we plan to add support for non-virtual subsystems and Stateflow. Stateflow is an implementation of Statecharts [17] in Simulink. To create simple reasoning rules, a mode-automata [21, 22] like structure of the diagrams will be used [8, 10]. Contracts for continuous and multi-rate systems would also be of interest.

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 [15], Ptolemy II [28] and Scicos [30] as well.

Acknowledgement

This work is carried out in the context of the project ITCEE (Improving Transient Control and Energy Efficiency by Digital Hydraulics) funded by TEKES (Finnish Funding Agency for Technology and Innovation)

References

[1] M. Abadi and L. Lamport. Conjoining specifications. *ACM Transactions on Programming Languages and Systems*, 17(3):507–534, 1995.

[2] J.-R. Abrial, E. Börger, and H. Langmaack. The steam boiler case study: Competition of formal program specification and development methods. In *Formal Methods for Industrial Applications - Specifying and Programming the Steam Boiler Control*, volume 1165 of *LNCS*, pages 1–12. Springer, 1996.

[3] R. Arthan, P. Caseley, C. O'Halloran, and A. Smith. ClawZ: Control laws in Z. In *Proceedings of ICFEM 2000*, pages 169–176. IEEE Press, 2000. `http://www.lemma-one.com/clawz_docs/clawz_docs.html`.

[4] R.-J. R. Back and R. Kurki-Suonio. Decentralization of process nets with centralized control. In *Proceedings of the 2nd ACM SIGACT-SIGOPS Symposium of Principles of Distributed Computing*, pages 131–142, 1983.

[5] R.-J. R. Back and J. von Wright. Trace refinement of action systems. In B. Jonsson and J. Parrow, editors, *Proc. of the 5th International Conference on Concurrency Theory, CONCUR'94*, pages 367–384, Uppsala, Sweden, 1994. Springer-Verlag.

[6] R.-J. R. Back and J. von Wright. *Refinement Calculus: A Systematic Introduction*. Graduate Texts in Computer Science. Springer-Verlag, 1998.

[7] R.-J. R. Back and J. von Wright. Compositional action system refinement. *Formal Aspects of Computing*, 15:103–117, 2003.

[8] P. Boström, M. Linjama, L. Morel, L. Siivonen, and M. Waldén. Design and validation of digital controllers for hydraulics systems. In *The 10th Scandinavian International Conference on Fluid Power*, volume 1, pages 227–241, Tampere, Finland, 2007.

[9] P. Boström, M. Linjama, L. Morel, L. Siivonen, and M. Waldén. Design and validation of digital controllers for hydraulics systems. Technical Report 800, Turku Centre for Computer Science, 2007.

[10] P. Boström and L. Morel. Mode-automata in Simulink/Stateflow. Technical Report 772, Turku Centre for Computer Science, 2006.

[11] L. Burdy, Y. Cheon, D. Cok, M. Ernst, J. Kiniry, G. T. Leavens, K. R. M. Leino, and E. Poll. An overview of JML tools and applications. *International Journal on Software Tools for Technology Transfer*, 7(3):212–232, 2005. `http://www.cs.iastate.edu/~leavens/JML/`.

[12] A. Cavalcanti, P. Clayton, and C. O'Halloran. Control law diagrams in Circus. In J. S. Fitzgerald, I. J. Hayes, and A. Tarlecki, editors, *Proceedings of FM 2005*, volume 3582 of *LNCS*, pages 253–268. Springer-Verlag, 2005.

[13] C. Chen and J. S. Dong. Applying timed interval calculus to Simulink diagrams. In *Eight International Conference on Formal Engineering Methods, ICFEM 2006*, volume 4260 of *LNCS*, pages 74–93, Macao, China, 2006. Springer-Verlag.

[14] E. W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, Englewood Cliffs, NJ, 1976.

[15] Esterel Technologies. SCADE. `http://www.esterel-technologies.com/products/scade-suite/`, 2006.

[16] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language Lustre. *Proceedings of the IEEE*, 79(9):1305–1320, 1991.

[17] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, 1987.

[18] I. J. Hayes, M. A. Jackson, and C. B. Jones. Determining the specification of a control system from that of its environment. In K. Araki, S. Gnesi, and D. Mandrioli, editors, *Proceedings of FME 2003: Formal methods*, volume 2805 of *LNCS*, pages 154–169. Springer-Verlag, 2003.

[19] B. Mahony. The DOVE approach to design of complex dynamic processes. In V. A. Carreño, C. A. Muñoz, and S. Tahar, editors, *Theorem Proving in Higher Order Logic*, volume CP-2002-211736 of *NASA conference publication*. NASA, 2002.

[20] F. Maraninchi and L. Morel. Logical-time contracts for reactive embedded components. In *30th EUROMICRO Conference on Component-Based Software Engineering Track, ECBSE'04*, Rennes, France, August 2004.

[21] F. Maraninchi and Y. Rémond. Mode-automata: About modes and states for reactive systems. In *European Symposium on Programming*, volume 1381 of *LNCS*. Springer Verlag, 1998.

[22] F. Maraninchi and Y. Rémond. Mode-automata: a new domain-specific construct for the development of safe critical systems. *Science of Computer Programming*, 46(3):219–254, 2003.

[23] Mathworks Inc. Simulink. `http://www.mathworks.com/products/simulink`, 2006.

[24] L. Meinicke and I. Hayes. Continuous action system refinement. In T. Uustalo, editor, *Proceedings of MPC 2006*, volume 4014 of *LNCS*, pages 316–337. Springer-Verlag, 2006.

[25] B. Meyer. *Object-Oriented Software Construction*. Prentice-Hall, 2 edition, 1997.

[26] J. Mikáč and P. Caspi. Temporal refinement for Lustre. In *Proceedings of Synchronous Languages, Applications and Programming, SLAP 2005*, ENTCS, Edinburgh, Scotland, 2005. Elsevier.

[27] J. Misra and K. M. Chandy. Proofs of networks of processes. *IEEE Transactions on Software Engineering*, 7(4):417–426, 1981.

[28] Ptolemy Project. Ptolemy II. `http://ptolemy.berkeley.edu/ptolemyII/`, 2005.

[29] M. Rönkkö, A. Ravn, and K. Sere. Refinement and continuous behaviour. In J. H. van Schuppen and F. W. Vaandrager, editors, *Hybrid Systems: Computation and Control, Second International Workshop HSCC'99*, volume 1569 of *LNCS*, pages 223–237. Springer-Verlag, 1999.

[30] Scilab Consortium. Scilab/Scicos. `http://www.scilab.org`, 2006.

[31] S. Skogestad and I. Postlethwaite. *Multivariable Feedback Control: Analysis and Design*. Wiley, 2 edition, 2005.

[32] K. Stølen. Assumption/commitment rules for dataflow networks - with emphasis on completeness. In H. Riis Nielson, editor, *Programming Languages and Systems - ESOP'96, 6th European Symposium on Programming*, volume 1058 of *LNCS*, pages 356–372. Springer, 1996.

[33] A. Tiwari, N. Shankar, and J. Rushby. Invisible formal methods for embedded control systems. *Proceedings of the IEEE*, 91(1):29–39, January 2003.

[34] S. Tripakis, C. Sofronis, P. Caspi, and A. Curic. Translating discrete-time Simulink to Lustre. *ACM Transactions on Embedded Computing Systems (TECS)*, 4(4):779–818, 2005.

Joukahaisenkatu 3-5, 20520 Turku, Finland | www.tucs.fi

University of Turku
- Department of Information Technology
- Department of Mathematics

Åbo Akademi University
- Department of Information Technologies

Turku School of Economics and Business Administration
- Institute of Information Systems Sciences

ISBN 978-952-12-1905-4
ISSN 1239-1891