ASM, Controller Synthesis, and Complete Refinement

Richard Banach[a,1], Huibiao Zhu[b,2], Wen Su[c], Xiaofeng Wu[b]

[a]*School of Computer Science, University of Manchester,*
Oxford Road, Manchester, M13 9PL, U.K.
[b]*Shanghai Key Laboratory of Trustworthy Computing, East China Normal University,*
3663 Zhongshan Road North, Shanghai 200062, P.R. China.
[c]*School of Computer Engineering and Science, Shanghai University,*
Shanghai, P.R. China.

Abstract

While many systems are naturally viewed as the interaction between a controller subsystem and a controlled, or plant subsystem, they are often most easily initially understood and designed monolithically, simply as a collection of variables that represent various aspects of the system, which interact in the most self-evident way. A practical implementation needs to separate controller from plant though. We study the problem of when a monolithic ASM system can be split into controller and plant subsystems along syntactic lines derived from variables' natural affiliations. We give restrictions that enable the split to be carried out cleanly, and we give conditions that ensure that the resulting pair of controller and plant subsystems have the same behaviours as the original design. We relate this phenomenon to the concept of *complete* refinement in ASM. Making this strategy work effectively, usually requires a nontrivial domain theory, into which a number of properties which are neither the sole possession of the controller subsystem nor of the plant subsystem must be placed. We argue that these properties are latent in the original monolithic model. We illustrate the theory with a case study concerning eating with chopsticks. This leads to an extension of controller synthesis for continuous ASM systems, which are briefly covered. The chopsticks case study is then extended into the continuous sphere.

1. Introduction

Today, when one considers the ubiquity of embedded controllers, which take on the digital role in the interaction of a digital and an external system, it becomes clear that many systems are naturally viewed as the interaction between a controller subsystem and a controlled, or plant subsystem. Regarding the high level design of such systems, the fact that the ultimate design needs to be split into controller and plant subsystems is evident from the outset. However, it is often easier in the earlier stages of design to ignore that fact, and to focus exclusively on the overall system goals. This means postponing for the time being the issue of how the solution arrived at is to be organised into the two subsystems. Such a *monolithic* approach means that there is simply less to worry about in the earlier stages of design, when there is the most uncertainty concerning the most critical aspects of the problem. This allows the bulk of this early design activity to focus on the overall goals rather than lower level technical detail.

Email addresses: banach@cs.man.ac.uk (Richard Banach), hbzhu@sei.ecnu.edu.cn (Huibiao Zhu), wsu@shu.edu.cn (Wen Su), xfwu@sei.ecnu.edu.cn (Xiaofeng Wu)

[1]A large portion of the work reported in this paper was done while the first author was a visiting researcher at the Shanghai Key Laboratory of Trustworthy Computing at East China Normal University. The support of ECNU is gratefully acknowledged.

[2]Huibiao Zhu is supported by National High Technology Research and Development Program of China (No. 2012AA011205), National Natural Science Foundation of China (No. 61361136002 and No. 61321064), Shanghai Knowledge Service Platform Project (No. ZF1213) and Shanghai Minhang Talent Project.

However, a practical implementation needs to separate the controller from the plant, since it is the controller which behaves according to a human-created digital design, and the plant behaves according to patterns determined by the laws of nature. In this paper we study the problem of when a monolithic ASM system design, embodying this dual controller/plant nature, can be split into separate controller and plant subsystems. This is to be done along generic syntactic lines derived from the most natural associations of the system variables to one or other (controller or plant) subsystem. The approach generalises a specific case study in which this task arose and where it was tackled rather informally [2]. We find that the success of the generic approach to such a goal requires that the monolithic design satisfies some simple criteria *ab initio*. As well as studying the problem from an abstract viewpoint, we present some examples.

In more detail, the rest of the paper is as follows. Section 2 describes the controller synthesis problem in abstract terms, focusing on the specific way that controller and plant are to be separated. A sufficient condition for the success of the desired controller/plant separation is formulated and proved. The undecidability of controller synthesis is also discussed in Section 2.1 by reduction to the Halting Problem. In Section 3 we consider a straightforward computable approximation to the controller synthesis problem, and argue that it is adequate for practical purposes. Section 4 discusses the role of the domain theory in the formulation of the controller synthesis problem — in many cases, the rules governing the behaviour of the system overall, can be viewed as belonging neither entirely to the controller subsystem nor entirely to the plant subsystem. Section 5 relates the preceding material to the ASM concept of *complete refinement*. When the controller synthesis problem is resolved successfully, each version of the overall system description refines the other. Section 6 introduces an example based on the idea of picking up food with chopsticks, viewed as a control problem. Section 7 extrapolates the preceding ideas to the case of continuous ASM, in which smoothly changing (as well as discretely changing) behaviours are admitted. Section 8 extends the discussion of the chopsticks case study by taking on board the continuous notions. In section 9, we loosen the tight synchronisation between controller and plant, evident in the account so far, to create a slightly more liberal framework for the continuous case. Section 10 concludes.

2. The Controller Synthesis Problem

We consider a generic ASM system consisting of basic ASM rules using straightforward single variable locations and a simple element of nondeterminism. Following [6], for our purposes, such a rule can be written as:

$$\begin{array}{l} \textsc{Op}(pars) \;= \\ \textbf{if}\ guard(xs, pars)\ \ \textbf{then choose}\ xs'\ \ \textbf{with}\ rel(xs', xs, pars)\ \ \textbf{do}\ xs \;:=\; xs' \end{array} \tag{1}$$

In (1), *pars* are the input parameters (as needed) and *xs* are the variables modified by the rule. The rule's guard is *guard*, and *rel* represents the relationship that is to hold between the parameters, the before-values of the variables *xs*, and their after-values referred to as xs', when the rule fires. As usual, in a single step of a run of the system, all rules which are enabled (i.e. whose guards are true) fire simultaneously, provided that the totality of updates defined thereby is consistent, else the run aborts.

In this paper we are interested in control applications, and we envisage the design done in a monolithic way at the outset, addressing system-wide design goals before plunging into the details of subsystem design. Thus the design may start by being expressed using system-wide variables. However, by a process of gradual refinement, the collection of variables will eventually end up such that each variable can be identified as belonging to either the controller-subsystem-to-be, or the plant-subsystem-to-be. Despite this prospective partition of the variables though, a typical legacy of the top-down design process will be that many, or even all, of the rules of the sytem description will still involve variables of both kinds.

The controller synthesis problem is the problem of taking such a collection of rules (call it *Sys*), and separating it into one set of rules for the controller (call it *Con*) and another set for the plant (call it *Pla*), such that each subsystem of rules reads only the variables accessible to it, and each modifies only the variables that it owns. Moreover, this is to be done in such a way that the combination of the rules in *Con* and *Pla* generates the same behaviour (i.e. the same set of runs) as the original ruleset *Sys*.

Note that in [6], the importance of distinguishing *controlled* functions from *monitored* ones is firmly stressed, in one sense solving the controller synthesis problem right at the outset, since the distinction already separates the controller from the plant. Our perspective is different however, since it permits this aspect to be postponed for an initial portion of the development. In this sense, the activity of *deriving which are the controlled and which are the monitored functions is brought under the umbrella of the formal development process* in our approach, since it permits some formal scrutiny of a stage of the development that otherwise would be done entirely informally. Our goal in the present work is to ask therefore under what conditions the separation can be done at a suitable moment in a systematic way.

We perform the separation in a systematic manner. We assume that the variables *Var* of *Sys* can be partitioned into the variables for which the controller has write access, written $xs_C \subseteq Var_C$, and the variables for which the plant has write access, written $xs_P \subseteq Var_P$; with $Var_C \cap Var_P = \varnothing$. We assume that for each rule $\mathrm{OP}(pars) \in Sys$, the guard can be written in the form $guard(xs, pars) \equiv guard_C(xs_C, xs_P^c, pars_C) \wedge guard_P(xs_P, xs_C^p, pars_P)$, where xs_P^c are the plant variables to which the controller has read access, and xs_C^p are the controller variables to which the plant has read access. We also assume that for each rule, the update relation $rel(xs', xs, pars)$ can be written in the form $rel(xs', xs, pars) \equiv rel_C(xs_C', xs_C, xs_P^c, pars_C) \wedge rel_P(xs_P', xs_P, xs_C^p, pars_P)$. We say that a system is **admissible** (with respect to the given method of splitting) iff the above hold. (We also call a system admissible when the method of splitting has not been explicitly described, but is hypothesised to exist — whereupon, if there is more than one such splitting, it is assumed that one particular one is borne in mind and is to remain fixed for the duration of the relevant discourse.[3])

We view the ease with which an admissible splitting can be achieved for a given system as a vindication of the appropriate and successful completion of the earlier stages of the development. In this regard, and especially since the standard approach advocated in [6] is predicated on resolving these matters *a priori*, we expect that arriving at an admissible splitting should be no more problematic than the invention of the initial system model would be in the conventional approach.

Under the above assumptions, the desired construction is relatively clear. For each rule like (1) in *Sys*, we generate two fresh rules:

$$\begin{array}{l} \mathrm{OP}_C(pars) \; = \\ \textbf{if} \;\; guard_C(xs_C, xs_P^c, pars_C) \;\; \textbf{then choose} \;\; xs_C' \;\; \textbf{with} \;\; rel_C(xs_C', xs_C, xs_P^c, pars_C) \\ \quad \textbf{do} \;\; xs_C \;:=\; xs_C' \end{array} \tag{2}$$

$$\begin{array}{l} \mathrm{OP}_P(pars) \; = \\ \textbf{if} \;\; guard_P(xs_P, xs_C^p, pars_P) \;\; \textbf{then choose} \;\; xs_P' \;\; \textbf{with} \;\; rel_P(xs_P', xs_P, xs_C^p, pars_P) \\ \quad \textbf{do} \;\; xs_P \;:=\; xs_P' \end{array} \tag{3}$$

Of these, (2), called the *C*-portion, goes into the controller subsystem *Con*, and (3), called the *P*-portion, goes into the plant subsystem *Pla*. We complete the construction of *Con* and *Pla* by defining their initial states. These are constructed by restricting the initial states of *Sys* to the variables in Var_C and Var_P respectively.

[3] In these circumstances, it does not matter *which* such splitting is intended, for, without further information, only remarks which are generically applicable to *all* such splittings can be made.

(Technically, the *Con*, and *Pla* initial states are generated by existentially quantifying out the variables $Var - Var_C$ of the *Sys* initial states in *Con*, and existentially quantifying out the variables $Var - Var_P$ of the *Sys* initial states in *Pla* — provided there are no non-trivial joint initial properties. If there are non-trivial joint initial properties, in other words if the set of initial states of *Sys* is *not* just the Cartesian product of the initial states of the *Con* variables and the initial states of the *Pla* variables, then the construction cannot be carried through.)

Let us now consider the system Sys_{C+P}, which consists of the variables and initial states of *Sys*, and whose rules are the union of the OP_C rules from *Con* and the OP_P rules from *Pla*. It is rather obvious that whenever a rule OP of *Sys* is enabled, in Sys_{C+P}, the corresponding rules OP_C and OP_P from Sys_C and Sys_P will also be enabled, since their guards are just weakenings of OP's guard. Consequently the runs of *Sys* are a subset of the runs of Sys_{C+P}

(Returning to the issue of initial states, technically, the initial states of Sys_{C+P} are recovered by conjoining initial state declarations of Sys_C and Sys_P. This raises the intriguing possibility that more states could be declared initial by doing this than are declared originally in *Sys*, since the existential quantifications in the initial declarations in Sys_C and Sys_P may include more states than are defined in the initial declaration of *Sys* itself. We confirm that the 'no non-trivial joint initial properties' stipulation, as defined above, prevents this, enabling the identification of the initial states of Sys_{C+P} with those of *Sys*.)

Regarding runs though, the runs of *Sys* may be a *proper* subset of the runs of Sys_{C+P} since the guards of the individual OP_C and OP_P rules are weaker than the guard of OP, and so, in certain states, may enable one or other of OP_C and OP_P without the counterpart rule being enabled. This is highly undesirable from a requirements point of view of course, since the overall objective was to achieve the behaviour of *Sys*, and not to introduce some spurious additional behaviours.

Definition 2.1. *A system Sys, with $Var = Var_C \uplus Var_P$ which is admissible, has a resolvable controller synthesis problem iff, after the construction above, there is a bijection κ between the runs of Sys and the runs of Sys_{C+P}, such that for each step of a run of Sys matched by κ to a run of Sys_{C+P}, the rules used in the Sys_{C+P} step are exactly the C-portions and P-portions of the rules used in the corresponding step of the Sys run.*

Next, we give a sufficient condition for resolvability of the controller synthesis problem. It features the domain theory for the development of the system *Sys*. The domain theory is intended as a repository for facts about the variables in the two subsystems that are needed to establish the equivalence of the original and partitioned systems. We elaborate the role of the domain theory much more extensively in Section 4.

Theorem 2.2. *Suppose a system Sys is admissible. Then Sys has a resolvable controller synthesis problem if:*

For all rules OP*, their derived rules* OP_C *and* OP_P*, and reachable states xs* •

$$[\ Domain(xs) \wedge guard_C(xs_C, xs_P^c, pars_C) \Rightarrow guard(xs, pars)\ ] \ \wedge$$
$$[\ Domain(xs) \wedge guard_P(xs_P, xs_C^p, pars_P) \Rightarrow guard(xs, pars)\ ] \qquad (4)$$

where $Domain(xs)$ is the domain theory for the development of Sys.

Proof: To get the result, it is sufficient to show that when (4) holds, every run of Sys_{C+P} amounts to a run of *Sys*, since we argued above that all *Sys* runs are Sys_{C+P} runs already (under an obvious injection of runs κ, that maps *Sys* runs to Sys_{C+P} runs by mapping each *Sys* step to the Sys_{C+P} step consisting of its *C*-portions and *P*-portions).

We proceed by induction on the length of the run. The base case is trivial since the initial states of *Sys* and of Sys_{C+P} are stipulated to be identical. Suppose then that we have the result for all Sys_{C+P} runs of

length n or less. Choose a run π of length n, reaching state xs, which is extendable. This means that there is some set of rules with a consistent update set in state xs, that is enabled in xs. Let OP_C, a C-portion rule, be one such rule (the argument is symmetrical for P-portion rules). Since OP_C is enabled in xs, its guard $guard_C$ holds, whence $guard$ holds by (4). Since $guard_P$ weakens $guard$, $guard_P$ holds, whence the corresponding P-portion, OP_P, is enabled. Since both OP_C and OP_P are enabled for every such rule, the update of Sys is emulated by Sys_{C+P} in the next step of the run. Doing the same for all possible ways of extending all extendable runs of length n completes the inductive step. This allows us to conclude that all runs of Sys_{C+P} correspond to runs of Sys in a way that extends κ to a bijection of the required kind. □

2.1. Undecidability of Controller Synthesis

The presence of the reachability criterion in (4) makes the following result relatively unsurprising.

Theorem 2.3. *The resolvability of the controller synthesis problem is undecidable.*

Proof: We outline a reduction of the Halting Problem to the controller synthesis problem. Let TM be an arbitrary Turing Machine. Let TM^0_C be an emulation of TM by an ASM constructed in a rather obvious way: i.e. there is an alphabet of states, another of tape symbols, a variable for the current state, a data structure for the tape, and a separate rule for each transition in the transition relation of TM. Let TM^0_P be another such ASM emulation, isomorphic to TM^0_C, but with all alphabets and variable names completely disjoint from those of TM^0_C. Therefore the following exist: a bijection between the (disjoint) alphabets of states of TM^0_C and TM^0_P; a bijection between the (disjoint) alphabets of tape symbols of TM^0_C and TM^0_P; a bijection between values of the current state variables of TM^0_C and TM^0_P; a bijection between the tape data structures of TM^0_C and TM^0_P; and a bijection between the sets of rules for the TM^0_C and TM^0_P versions of the transitions of TM that work with respect to the preceding collection of bijections.

Now consider the ASM TM^0_{C+P} constructed as in the previous section. It has twice as many rules as TM has transitions, but due to the bijections mentioned, they are enabled pairwise at exactly the same moments, so TM^0_{C+P} just emulates two disjoint copies of TM running in lockstep.

By contrast, consider the ASM $TM^0_{C\wedge P}$ constructed as follows. We fuse each pair of rules of TM^0_{C+P} that correspond via the bijections, into a single rule, by conjoining the two guards, and combining the updates. The Turing Machine $TM^0_{C\wedge P}$ has exactly as many rules as TM has transitions since the doubling in TM^0_{C+P} has been removed. It is clear that $TM^0_{C\wedge P}$ and TM^0_{C+P} are strongly bisimilar to each other, and to TM, by a simple inductive argument over the length of runs of these ASMs and of TM. Now we modify TM^0_C, and modify TM^0_P, as follows.

Since TM is arbitrary, it may contain halting before-configs —i.e. pairs (t,s) where t is a tape symbol and s is a state— from which no transition issues. If TM has a halting before-config (t,s), we do the following. Let (t_C,s_C) be the counterpart of (t,s) in TM^0_C. To TM^0_C we add a rule that implements a self-loop guarded on (t_C,s_C) (without moving the tape head). Call the result of this modification TM_C. Let (t_P,s_P) be the counterpart of (t,s) in TM^0_P. To TM^0_P we add a rule that implements a self-loop guarded on s_P alone (i.e. ignoring the tape symbol, and without moving the tape head), calling this modification TM_P.

Now consider the two ASM systems $TM_{C\wedge P}$ and TM_{C+P}, created from TM_C and TM_P by manipulations analogous to the ones that produced TM^0_{C+P} and $TM^0_{C\wedge P}$ from TM^0_C and TM^0_P. In $TM_{C\wedge P}$ (which plays the role of Sys above), the stronger guard of the TM_C rule, in effect subsumes the weaker one of the TM_P rule, and the fused rule is only enabled exactly when the TM_C rule is enabled. However in TM_{C+P} (which plays the role of Sys_{C+P} above), this is not the case. There, the TM_P rule exists independently, and if the computation of TM reaches a machine configuration in which the state is s, then the TM_P rule is enabled when the tape symbol and state are $(\tilde{t},s)$, for any $\tilde{t} \neq t$ (as well as when the tape symbol is t), giving rise to observable behaviours not reflected in $TM_{C\wedge P}$.

Since (the rules defining) the behaviour at the halting before-config (t,s) are the only thing we have changed, we can say that if a before-config $(\tilde{t},s)$ (for any $\tilde{t}$) is *never* reached during the computation of *TM*, then $TM_{C\wedge P}$ and TM_{C+P} both reflect the behaviour of *TM*, and thus stay in lockstep, and that therefore $TM_{C\wedge P}$ has a resolvable controller synthesis problem when it is resolved into TM_C and TM_P and these are subsequently recombined into TM_{C+P}. On the other hand, if a before-config $(\tilde{t},s)$ (for some $\tilde{t}$) *is* reached during the computation of *TM*, then the behaviour of $TM_{C\wedge P}$ and TM_{C+P} after the first occurrence of such a before-config differ, and therefore $TM_{C\wedge P}$ does not have a resolvable controller synthesis problem when it is resolved into TM_C and TM_P and these are subsequently recombined into TM_{C+P}. By the undecidability of the Halting Problem, we cannot in general determine algorithmically whether a given halting before-config $(\tilde{t},s)$ is ever reached, so we cannot in turn algorithmically determine in general whether the controller synthesis problem is resolvable or not. □

3. Computable Controller Synthesis

Restricting to a safe approximation to reachability, we get a computable version of (4), which we argue will be adequate for all practical purposes.

Theorem 3.1. *Suppose a system Sys is admissible and XS is a set of states that includes all reachable states. Then Sys has a resolvable controller synthesis problem if:*

$$\begin{aligned}&\textit{For all rules } \textsc{Op}, \textit{ their derived rules } \textsc{Op}_C \textit{ and } \textsc{Op}_P, \textit{ and all } xs \in XS \bullet \\ &\quad [\ \mathit{Domain}(xs) \wedge \mathit{guard}_C(xs_C, xs_P^c, pars_C) \vdash \mathit{guard}(xs, pars)\ ]\ \wedge \\ &\quad [\ \mathit{Domain}(xs) \wedge \mathit{guard}_P(xs_P, xs_C^p, pars_P) \vdash \mathit{guard}(xs, pars)\ ] \qquad (5)\end{aligned}$$

where $\mathit{Domain}(xs)$ is the domain theory for Sys and $\vdash$ is provability in a suitable system.

4. The Role of the Domain Theory

In equations (4) and (5) we saw the presence of a domain theory $\mathit{Domain}(xs)$ underpinning the derivability of the whole-system guard from the partial guards of the controller and plant subrules $\textsc{Op}_C$ and $\textsc{Op}_P$ of a given whole-system rule Op. In this section we comment on this further.

In any formal development/verification system there will be a collection of definitions, of constants, static mathematical objects etc., that create a context for the remainder of the development. In ASM, these entities will be captured by static rather than dynamic functions. Often, when discussing the formal development/verification environment informally, one will not always mention this static context explicitly whenever it might be needed, but it will nevertheless still need to be present (for example among the hypotheses of a verification condition), otherwise needed properties of the identifiers occurring in a particular system model would be unavailable. What we mean by a domain theory in this paper, is an extension of this basic idea of a collection of axioms that support the remainder of the development.

Turning to the design process, one of the most natural consequences of early-phase monolithic design is that all sorts of issues can get entangled from the beginning. This in itself is no bad thing, as we have already said above, since it allows early-phase efforts to focus on the crucial application level issues rather than on technicalities of structure, but it does make the subsequent disentanglement rather more challenging than it otherwise might be.

When we contemplate disentangling an integrated design into controller and plant, certain aspects will naturally fall into the controller subsystem and others will naturally fall into the plant subsystem. Thus, when we partition the variables during our process of controller synthesis, each variable goes into either the *Con* or the *Pla* subsystem. However, there will typically be remaining parts of the monolithic design where it is not immediately obvious how to handle the allocation to one or other subsystem. Thus

there will be properties of the overall design that mix the variables of the two subsystems. Frequently, such aspects concern what might be referred to as elements of physical law that couple the behaviour of controller and plant variables. After all, it is physical law of this kind that we rely on when we envisage being able to adequately control the plant using the controller in the first place, given that they are, as we describe, two separate (though coupled) systems. Such elements of the monolithic design, expressing the unavoidable interdependence between the variables of the two subsystems, are the prime candidates for inclusion in the domain theory.

A more crude, though rather effective way of putting it, is that once the variables have been partitioned, and the rules and static properties of the original monolithic design that exclusively concern the variables of one or other subsystem have been allocated to that subsystem, everything else goes into the domain theory.

To illustrate with an example, suppose we have a system containing a massive body, whose motion we want to influence by the application of a force. The thing that determines the force to be applied to the body is Newton's Second Law of Motion, namely that acceleration is proportional to the force applied. In an integrated design, the Second Law would just be one of the equations that contributed to the specification of the solution, and would be treated the same way as all the other equations contributing to the design. However, in a design separated into controller and plant subsystems, one would have to decide how to express the Second Law and where to put it.

We face two issues. One issue is that the Second Law is not something that applies exclusively to the design being undertaken, but is much more generic. In that sense it does not 'belong' to the variables to which it is applied, but is a much more widely applicable phenomenon of nature. The other issue is whether the massive body and force both reside in one of the two subsystems into which we split our monolithic design, or not, with, in the latter case, the force in the controller typically, and the body in the plant. We address these in turn.

Regarding the first issue, the great general applicability of Newton's Second Law suggests a highly generic formulation. Newton's Second Law conventionally reads $F = m\ddot{x}$. This contains the free identifiers F, m and x — as well as globally understood constants for equality, application of the differentiation operation with respect to time, and (implicitly) multiplication. Taking it for granted that the globally understood constants are part of the fabric of the formalism, the identifiers F, m and x cannot, though, be taken as free in the normal formal sense. They stand for 'typical' names of force, mass and position in 'typical' informal descriptions of physical processes. Therefore, formally, they must be understood as bound variables to be substituted with the actual variables pertaining to a given problem. Using fairly conventional lambda notation, Newton's Second Law can become: $NSL \mathrel{\widehat{=}} \lambda\Phi.\lambda\mu.\lambda\xi.(\Phi = \mu\mathcal{D}.\mathcal{D}.\xi)$, where $\mathcal{D}$ is a formal symbol for differentiation with respect to time. The ideal place for such a generic expression of Newton's Second Law would be in a shared domain theory, from where it could be applied by any subsystem that needed it.

To utilise the *NSL* form of the Second Law in a specific application, we would apply *NSL* to the actual system variables which were subject to the Second Law. If these were F, m, a respectively, we would instantiate the bound variables of *NSL* thus: $NSL.F.m.a$. This (or an equivalent formal mechanism) would appropriately express: (a) the generality of Newton's Second Law, (b) its application to a specific example, and the relationship between these ideas.

Although the preceding gives an account of an 'ideal' method of formally incorporating generally applicable laws in a specific application, a couple of factors militate against following this process to the letter. Firstly, there is the loss of clarity deriving from the use of multiple identifiers for essentially the same thing, and the mechanisms of binding of free variables and instantiation of bound variables that manage this — a development using this technique is bound to be more obscure than one which avoids its use. Secondly, if mechanical reasoning (in any form) is to be used to support the development, then having to manage the abstraction/application mechanisms explicitly will normally dramatically impact

the power of any such reasoning system in a negative way, to the detriment of the overall development.

It is thus recommended that a less 'purist' approach is used in practice in most cases, which brings us to our second issue. Continuing with Newton's Second Law, such a less purist approach would embed an occurrence of the Second Law directly into the subsystem containing the variable being controlled by it, where the law would be written directly using the variables involved, e.g. $F = m\mathcal{D}\mathcal{D}x$. In effect, the previous application of F, m, a to *NSL* would be done informally.

If all the variables involved belonged to one or other subsystem, then there would be nothing more to say. However, typically, the plant will contain the controlled variable x, but the controller will contain the controlling force F. In that case the plant subsystem will contain the rule expressing $F = m\mathcal{D}\mathcal{D}x$ and it must be the case that the force variable can be read by the plant subsystem. In this case, the role of the domain theory would be reduced somewhat, since it would not need to contain the generic *NSL* statement.

Still, in situations as just described, there is often a user/requirements-led motivation for each subsystem to name the variables that it uses according to its own conventions. If that were the case, then an additional role for the domain theory would be to contain the equalities that connect differently named instances of the same overall system variable in the two subsystems to each other. (Such a state of affairs would also reduce the need for each subsystem to read variables of the other subsystem directly, as in (4) or (5), although the needed access would now be via the domain theory.)

Summarising, we have identified a range of roles for the domain theory in the context of controller synthesis, beyond merely holding the static context of the application. They range from answering the question: *Where do you put the (generic, or inter-subsystem) physics?*, to holding cross-cutting properties that interrelate variables of the two subsystems, to holding the gluing relationships that connect differently named versions of the same entity in subsystems adhering to their own internal naming conventions. Moreover, the points that we have discussed are widely applicable beyond ASM to model based approaches in general, since we have said practically nothing that was ASM-specific.

Nevertheless, there is one further point that *is* ASM-specific, to which we now turn. Consider a rule OP of the undecomposed system whose guard might be written $guard \equiv guard_C \wedge guard_P$, where $guard_C$ and $guard_P$ are the guards of the decomposed rules OP_C and OP_P. In the undecomposed case, ASM scheduling policy demands that only when both $guard_C$ and $guard_P$ are true, does the rule OP fire (since both together are equivalent to *guard*). In the decomposed case, therefore, to gain the equivalent behaviour, i.e. that the firing of either of OP_C or OP_P implies also the simultaneous firing of the other, we would need that each of $guard_C$ and $guard_P$ implies the other, too. Now, it might be that the simultaneous truth of $guard_C$ and $guard_P$ could follow from the truth of one of them alone, but it is highly unlikely that real designers would create a high level system design containing such a level of redundancy. Therefore, in general, each of $guard_C$ and $guard_P$ would, alone, be too weak to enable us to deduce *guard* (and thus the enabledness of OP) when only one of them held. Consequently, in order to able be to derive *guard* when only one of $guard_C$ or $guard_P$ is available, we would be forced, in general, to rely on some additional information.

It becomes an additional duty of the domain theory that we have been discussing, to supply the additional information needed. Normally, this additional information will be a consequence of the properties of the earlier monolithic design anyway, since our controller synthesis strategy amounts, in the end, to a kind of syntactic rearrangement, and thus should not introduce new semantic properties. However, the conjunction of the $guard_C$ and $guard_P$ guards in the monolithic rule, may conveniently obscure the underlying physical reason why neither part of the rule can fire without the other, and in the separation of the two subsystems, these reasons may need to be brought out more clearly in the properties contained in the domain theory. So, as well as its job of expressing the static context of the application, and its potential to act as repository for facts at the interface of controller and plant, the facts that the domain theory contains should be designed in such a way that the domain theory can succeed in bridging the gap

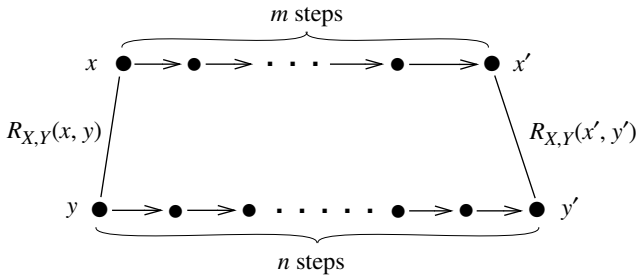

Figure 1: An ASM (m,n) diagram, showing how m abstract steps, going from state x to state x' simulate n concrete steps, going from y to y'. The simulation is embodied in the equivalence $R_{X,Y}$, which holds for the before-states of the series of steps $R_{X,Y}(x,y)$, and is re-established for the after-states of the series $R_{X,Y}(x',y')$.

between either of $guard_C$ or $guard_P$ alone, and their conjunction, in the way indicated in (4) and (5).

Like any formal derivation challenge, bridging the indicated gap can itself be relatively easy, or potentially difficult. In the worst case, it may require a detailed reachability argument, established by an induction over run length. However, the contents of the domain theory should be construed, at least in part, with the aim of helping to make this goal tractable. We have observed earlier that, in principle, the domain theory should not contain anything that is not a consequence of the original monolithic system. But this does not prevent it from explicitly mentioning less obvious consequences of these foundations as derived theorems, so that they may be more conveniently used to ease the proof of (4) and (5).

5. ASM Complete Refinement

In this section we explore the connection between the resolvability of the controller synthesis problem and the ASM concept of complete refinement. First we briefly review the necessary technical machinery.

In general, to prove that a concrete ASM system Y is a refinement of an abstract ASM system X, we verify so-called (m,n) diagrams, in which m abstract steps simulate n concrete ones in an appropriate way. The situation is illustrated in Fig. 1. It will be sufficient for us to focus on the refinement proof obligations (POs) which are the embodiment of this policy.

In Fig. 1 the equivalence,[4] $R_{X,Y}$, between abstract and concrete states, holds at the beginning and end of the (m,n) pair. This permits us to 'glue together' such (m,n) diagrams to create relationships between abstract and concrete runs in which $R_{X,Y}$ is periodically re-established. For our purposes, it will be sufficient to restrict to the $(1,1)$ case.

The first PO is the initialization PO:

$$\forall\, y \bullet YInit(y) \Rightarrow (\exists\, x \bullet XInit(x) \wedge R_{X,Y}(x,y)) \tag{6}$$

which demands that for every concrete initial state y there is an R-related abstract initial state x.

The second PO is correctness. The PO is concerned with the verification of (m,n) diagrams. In the general case, we would have to have some way of deciding which (m,n) diagrams are sufficient for the application in question, a problem that would often require an application-specific solution. However in the simpler $(1,1)$ case the solution is much more generic, amounting to straightforward $(1,1)$ simulation of all concrete steps, expressed by the following correctness PO:

$$\forall\, x,y,y' \bullet R_{X,Y}(x,y) \wedge \mathrm{YOP}(y,y') \Rightarrow (\exists\, x' \bullet \mathrm{XOP}(x,x') \wedge R_{X,Y}(x',y')) \tag{7}$$

[4] For the purposes of this paper, it is sufficient that the equivalence is understood to be a bijection.

In (7), it is demanded that whenever there is a concrete step $\mathrm{YOP}(y,y')$ carried out by a concrete operation YOP (where, by an operation, we mean a maximal enabled set of rules — provided its updates are consistent), and $R_{X,Y}(x,y)$ holds in the before-state, then an abstract step $\mathrm{XOP}(x,x')$ can be found to re-establish the equivalence $R_{X,Y}(x',y')$.

The ASM refinement policy also demands that non-termination be preserved from concrete to abstract runs. (The examples in this paper will not need this though.)

Assuming that (6) holds, and that we can prove (7) for *each* concrete step $\mathrm{YOP}(y,y')$, then the concrete model is a **correct** $(1,1)$ **refinement** of the abstract model. A correct refinement ensures that all functional properties of the concrete system, as seen through the equivalence *R*, are suitably reflected as properties of the abstract system. This is because of the direction of the implication in (7).

If we have a correct $(1,1)$ refinement, and in addition, the abstract system is also a correct refinement of the concrete system using the converse of the same equivalence *R*, then we have a **complete refinement** (of the abstract system by the concrete system). A complete refinement corresponds, in our model based world, to what is termed a strong bisimulation (through the state equivalence and input and output relations and their converses) in more abstract terminology. In a complete refinement, we can reverse the direction of the argument about preservation of properties, and state that the functional properties of the abstract system are preserved by the concrete system.

We return now to the controller synthesis problem, and show that resolvable controller synthesis coincides with a special case of complete refinement.

Theorem 5.1. *Let Sys be a system of ASM rules, and suppose that Sys is decomposed, according to a suitable partition of the variables, into two systems of rules Con and Pla, as described above. Let* Sys_{C+P} *be the recombined system. Then Sys has a resolvable controller synthesis problem with respect to the decomposition given, iff* Sys_{C+P} *is a complete refinement of Sys with respect to:*

(i) the equivalence on states defined by the identity restricted to reachable states, and,

(ii) the set of $(1,1)$ *diagrams given by:*

(a) relating each single Sys rule OP *to the simultaneous execution of its C-portion* OP_C *and its P-portion* OP_P*, and,*

(b) generating all the $(1,1)$ *diagrams* $(\mathrm{XOP},\mathrm{YOP})$ *derivable therefrom by the simultaneous scheduling (when simultaneously enabled), of maximal sets of Sys rules and their corresponding C- and P- portions — i.e. if* XOP *is a transition of* $\mathrm{OP}_1 \ldots \mathrm{OP}_n$*, this being a maximal set of simultaneously enabled Sys rules in some state of Sys, then the corresponding* YOP *is the corresponding transition of* $\mathrm{OP}_{C,1}, \mathrm{OP}_{P,1} \ldots \mathrm{OP}_{C,n}, \mathrm{OP}_{P,n}$.

In (i) of Theorem 5.1 we restrict the equivalence to reachable states in order that the weaker guards of the Sys_{C+P} rules do not enable them to fire at inopportune places in the state space that do not correspond to reachable states of *Sys*.

Proof: Suppose *Sys* has a resolvable controller synthesis problem with respect to the given decomposition into *Con* and *Pla*. We must show that Sys_{C+P} is a complete refinement of *Sys* with respect to the identity equivalence on reachable states and the set of $(1,1)$ diagrams given by refining a single *Sys* rule to the simultaneous execution of its *C*-portion and *P*-portion (and operations generated thereby).

To do this we must firstly show that all instances of the initialization PO (6), hold as given, and that they also hold when the roles of abstract (*Sys*) and concrete (Sys_{C+P}) systems are reversed. However, since the sets of initial states of *Sys* and Sys_{C+P} are identical by construction, and the equivalence *R* is an identity, this is essentially trivial.

Secondly, we must show that all instances of the correctness PO (7), hold as given for abstract *Sys* and concrete Sys_{C+P} systems, and that they also hold with the roles of abstract and concrete reversed. Let y be a concrete (i.e. Sys_{C+P}) state, reachable from some initial state via some run π. Let x be another name for y (allowing us to conveniently regard x as an abstract (i.e. *Sys*) state, also reachable from the same initial state via the counterpart of π matched by the κ of Definition 2.1). Since $x = y$, we have $R(x,y)$ where R is equality on reachable states. Suppose Sys_{C+P} makes a step $\mathrm{COP}_{C+P}(y,y')$. Then, since *Sys* has a resolvable controller synthesis problem, $\mathrm{COP}_{C+P}(y,y')$ consists of pairs of C-portions and P-portions of *Sys* rules, each pair derived by decomposing a single *Sys* rule. Therefore, the set of corresponding *Sys* rules can also make an abstract step $\mathrm{AOP}(x,x')$, where AOP consists of the *Sys* rules mentioned, and $x' = y'$ (so that $R(x',y')$). This establishes (7), showing that the AOP step simulates the COP_{C+P} step. Since R is an identity, and the steps of Sys_{C+P} (where *Sys* has a resolvable controller synthesis problem) are always performed by sets of the C-portions and P-portions of *Sys* rules, it is easy to invert this argument to show that for every such AOP step, the corresponding COP_{C+P} step simulates it too. In this manner, aggregating over all reachable states and all steps issuing from them, gives us the complete refinement required.

For the converse, suppose that Sys_{C+P} is a complete refinement of *Sys* with respect to the identity equivalence on reachable states, and the set of $(1,1)$ diagrams given by refining a single *Sys* rule to the pair of its C-portion and P-portion (and the set of operations that this generates). We must show that *Sys* has a resolvable controller synthesis problem with respect to the given decomposition into *Con* and *Pla*.

We proceed by induction on the length of runs. The base case is trivial since the initial states of *Sys* and of Sys_{C+P} are stipulated to be identical. So *Sys* and Sys_{C+P} have the same set of runs of length zero.

For the inductive step, we assume that the set of runs of *Sys* of length n or less, is in bijective correspondence with the set of runs of Sys_{C+P} of length n or less, via a bijection κ_n, in which corresponding steps of κ_n-related runs are performed by a set of rules in the *Sys* case, and exactly the set of C-portions and P-portions of the same set of rules in the Sys_{C+P} case. To go to $n+1$, consider one such run π of length n of *Sys* which is extendable. If a *Sys* step extends π, it is easy to see that, since we have a *complete* refinement, the *Sys* step can be simulated by a Sys_{C+P} step that splits each rule in the *Sys* step into its C-portion and its P-portion. Equally, since we have a refinement, and this covers *all* steps of Sys_{C+P} by definition, all steps of Sys_{C+P} are performed by sets of pairs of C-portions and P-portions of *Sys* rules (because of the way that the $(1,1)$ diagrams of the refinement are defined), and therefore we easily see that any Sys_{C+P} step that extends the κ_n image of a *Sys* run π, will be simulated in the obvious way by a *Sys* step that recombines all the C-portions and P-portions of the Sys_{C+P} step. This extends κ_n to this pair of extended runs. Doing the same for all possible extensions of all length n runs completes the inductive step. Ultimately we arrive at the required bijection between all runs of *Sys* and those of Sys_{C+P}. □

6. An Example: Eating with Chopsticks

We now look at a simple example of the preceding theory: eating food with chopsticks. To keep things simple, we do a statics based treatment of the problem, neglecting many aspects that would make it more realistic. For example, we neglect the role of gravity which obviously plays a part in genuine situations. Likewise, we ignore the role of the moments of the forces that we do consider round the fulcrum point of application on the chopsticks, viewing the problem as if all the forces were applied at a single point in order to simplify the calculations (the moments must balance of course). In this simplified framework, Fig. 2 shows the forces involved in grasping a morsel of food with chopsticks.

6.1. Food and Chopsticks

In a statically stable situation, the chopsticks extert forces on the food, and the food exerts equal and opposite forces on the chopsticks. The forces exerted by the food are $\boldsymbol{f}_{\mathrm{FU}}$ on the upper chopstick and

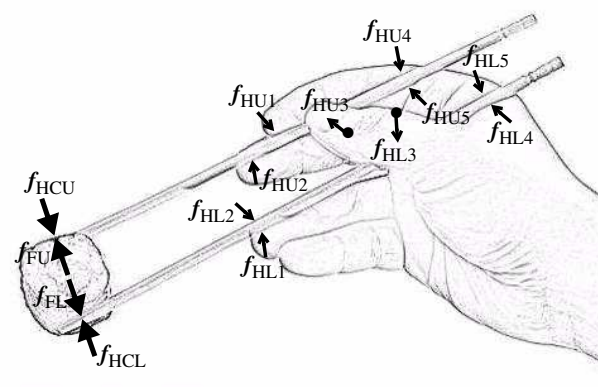

Figure 2: Forces involved in grasping a piece of food with chopsticks.

$\boldsymbol{f}_{\mathrm{FL}}$ on the lower chopstick. For simplicity we assume that these forces sum to zero (else the food would accelerate) and are also colinear.[5] Reacting to $\boldsymbol{f}_{\mathrm{FU}}$ and $\boldsymbol{f}_{\mathrm{FL}}$, the chopsticks exert their forces $\boldsymbol{f}_{\mathrm{HCU}}$ and $\boldsymbol{f}_{\mathrm{HCL}}$, equal and opposite to $\boldsymbol{f}_{\mathrm{FU}}$ and $\boldsymbol{f}_{\mathrm{FL}}$. So we have:

$$\boldsymbol{f}_{\mathrm{FU}} + \boldsymbol{f}_{\mathrm{FL}} = \mathbf{0} \tag{8}$$

$$\boldsymbol{f}_{\mathrm{HCU}} + \boldsymbol{f}_{\mathrm{HCL}} = \mathbf{0} \tag{9}$$

$$\boldsymbol{f}_{\mathrm{FU}} + \boldsymbol{f}_{\mathrm{HCU}} = \mathbf{0} \tag{10}$$

$$\boldsymbol{f}_{\mathrm{FL}} + \boldsymbol{f}_{\mathrm{HCL}} = \mathbf{0} \tag{11}$$

$$|\boldsymbol{f}_{\mathrm{FU}}| = |\boldsymbol{f}_{\mathrm{FL}}| = |\boldsymbol{f}_{\mathrm{HCU}}| = |\boldsymbol{f}_{\mathrm{HCL}}| \geq D \tag{12}$$

The last of these (12), expresses a constraint that the forces that we have mentioned in this problem have to be large enough (D) that they generate additional frictional forces (which can be taken to be proportional to the forces mentioned), which are sufficient to counteract gravity (which we have not taken into account), and which thereby stop the food from dislodging from the chopsticks when lifted.

We can write this as an ASM model, with a rule:

$$\begin{array}{l}\textsc{GraspFood} = \\ \textbf{choose } \boldsymbol{f}'_{\mathrm{FU}}, \boldsymbol{f}'_{\mathrm{FL}}, \boldsymbol{f}'_{\mathrm{HCU}}, \boldsymbol{f}'_{\mathrm{HCL}} \\ \quad \textbf{with } \boldsymbol{f}'_{\mathrm{FU}} + \boldsymbol{f}'_{\mathrm{FL}} = \boldsymbol{f}'_{\mathrm{HCU}} + \boldsymbol{f}'_{\mathrm{HCL}} = \boldsymbol{f}'_{\mathrm{FU}} + \boldsymbol{f}'_{\mathrm{HCU}} = \boldsymbol{f}'_{\mathrm{FL}} + \boldsymbol{f}'_{\mathrm{HCL}} = \mathbf{0} \wedge \\ \qquad |\boldsymbol{f}'_{\mathrm{FU}}| = |\boldsymbol{f}'_{\mathrm{FL}}| = |\boldsymbol{f}'_{\mathrm{HCU}}| = |\boldsymbol{f}'_{\mathrm{HCL}}| \geq D \\ \quad \textbf{do } \boldsymbol{f}_{\mathrm{FU}} := \boldsymbol{f}'_{\mathrm{FU}}, \boldsymbol{f}_{\mathrm{FL}} := \boldsymbol{f}'_{\mathrm{FL}}, \boldsymbol{f}_{\mathrm{HCU}} := \boldsymbol{f}'_{\mathrm{HCU}}, \boldsymbol{f}_{\mathrm{HCL}} := \boldsymbol{f}'_{\mathrm{HCL}}, \\ \qquad \mathit{grasped} := \textsc{true}\end{array} \tag{13}$$

There will be another rule DISLODGEFOOD, differing from (13) in the replacement of '$\geq D$' by '$< D$' and of TRUE by FALSE, which models the dislodgement of food as being due to the application of inadequate force, and disregarding any other maladroitness on the part of the user. Given the similarity of the two rules, we will not mention DISLODGEFOOD further, unless it is unavoidable, in order to avoid clutter.

We can regard GRASPFOOD (and DISLODGEFOOD) as a simple design for a control system — the chopsticks are intended to control the food by grasping it. Thus we can pursue our earlier strategy by

[5] In reality, slight deviations from colinearity are compensated for by forces of friction and deformation arising from the food, aided where appropriate, by surface tension forces coming from any sauce that the food might be prepared in (as well as many other similar considerations which we neglect).

separating the system into plant (food) and controller (chopsticks) subsystems. The GRASPFOOD rule separates into GRASPFOOD_C and GRASPFOOD_P:

$$\begin{array}{l}\text{GRASPFOOD}_C = \\ \textbf{choose } \boldsymbol{f}'_{\text{HCU}}, \boldsymbol{f}'_{\text{HCL}} \\ \quad \textbf{with } \boldsymbol{f}'_{\text{HCU}} + \boldsymbol{f}'_{\text{HCL}} = \mathbf{0} \wedge |\boldsymbol{f}'_{\text{HCU}}| = |\boldsymbol{f}'_{\text{HCL}}| \geq D \\ \quad \textbf{do } \boldsymbol{f}_{\text{HCU}} := \boldsymbol{f}'_{\text{HCU}}, \boldsymbol{f}_{\text{HCL}} := \boldsymbol{f}'_{\text{HCL}}, \\ \qquad \textit{grasped} := \text{TRUE}\end{array} \tag{14}$$

$$\begin{array}{l}\text{GRASPFOOD}_P = \\ \textbf{choose } \boldsymbol{f}'_{\text{FU}}, \boldsymbol{f}'_{\text{FL}} \\ \quad \textbf{with } \boldsymbol{f}'_{\text{FU}} + \boldsymbol{f}'_{\text{FL}} = \mathbf{0} \\ \quad \textbf{do } \boldsymbol{f}_{\text{FU}} := \boldsymbol{f}'_{\text{FU}}, \boldsymbol{f}_{\text{FL}} := \boldsymbol{f}'_{\text{FL}}\end{array} \tag{15}$$

In (14) and (15) we see that GRASPFOOD_C only 'owns' $\boldsymbol{f}_{\text{HCU}}$ and $\boldsymbol{f}_{\text{HCL}}$, so only assigns to those variables, and GRASPFOOD_P only 'owns' $\boldsymbol{f}_{\text{FU}}$ and $\boldsymbol{f}_{\text{FL}}$, so only assigns to them. We also observe that some pieces of GRASPFOOD are not present in either GRASPFOOD_C or GRASPFOOD_P, namely the terms that relate the food forces to the chopstick forces. This is explained by the observation that the relevant equations are part of the domain theory of statics: action and reaction are *always* equal statically, by Newton's Third Law. (This is an example of our comments in Section 4, with the domain theory playing the role of 'somewhere to put the physics'.) Additionally, the statement that successful grasping needs adequate force is also part of the domain theory, so we can write the domain as:

$$\textit{Domain}_{FHC} \equiv \boldsymbol{f}_{\text{FU}} + \boldsymbol{f}_{\text{HCU}} = \mathbf{0} \wedge \boldsymbol{f}_{\text{FL}} + \boldsymbol{f}_{\text{HCL}} = \mathbf{0} \wedge (\textit{grasped} = \text{TRUE} \Leftrightarrow |\boldsymbol{f}_{\text{HCL}}| \geq D) \tag{16}$$

Now, in the context of (16), it is easy to see that:

$$\textit{Domain}_{FHC} \wedge \textit{guard}_{\text{GRASPFOOD}_C} \vdash \textit{guard}_{\text{GRASPFOOD}} \tag{17}$$

$$\textit{Domain}_{FHC} \wedge \textit{guard}_{\text{GRASPFOOD}_P} \vdash \textit{guard}_{\text{GRASPFOOD}} \tag{18}$$

6.2. Chopsticks and Hand

The preceding was rather elementary. In particular, it presumed that chopsticks somehow grasp food by themselves, which is silly. In reality, chopsticks are held in the right hand, which causes them to exert the forces spoken of previously. We can now enrich our model by considering the hand-chopstick system as a further control system, and decomposing it further into a plant subsystem (the chopsticks themselves) and a controller subsystem (the hand).

We refer to Fig. 2 again. For a solid object to remain stable in 3D space, it needs to have four non-colinear forces summing to zero acting on it. If gravity is acting (as it normally is) then it supplies one force, and we derive the well-known fact that an object needs to be supported from underneath by three or more forces for stability.

This applies to the hand-chopstick system, where for simplicity, we are ignoring gravity. Given how chopstick are disposed with respect to the hand, it is in fact convenient to view the hand as exerting five forces per chopstick. Fig. 2 shows the forces involved.

The middle of the lower chopstick is held steady on the ring finger. Typically it is gently wedged in the angle between the edge of the fingernail and the side of the fleshy pad of the fingertip, which we model by the forces $\boldsymbol{f}_{\text{HL1}}$ and $\boldsymbol{f}_{\text{HL2}}$ in Fig. 2. These are predominantly directed in the plane of the diagram, with a small component at right angles, out of the plane of the diagram, towards the reader. The back end of the lower chopstick is held on the fleshy part between the thumb and palm, and the forces are modeled by $\boldsymbol{f}_{\text{HL4}}$ and $\boldsymbol{f}_{\text{HL5}}$. Again these are mostly in the plane of the diagram, with a small component outwards, towards the reader. Opposing all the outwards components is $\boldsymbol{f}_{\text{HL3}}$ (the force drawn with the

blob at its tail in Fig. 2), which is exerted by the lower end of the thumb, predominantly inwards into the diagram.[6]

If the chopstick is merely being held steady, then these forces sum to zero. However, if food is being held, then the user adjusts the individual forces so that they sum to $\boldsymbol{f}_{\mathrm{HCL}}$:

$$\boldsymbol{f}_{\mathrm{HL1}} + \boldsymbol{f}_{\mathrm{HL2}} + \boldsymbol{f}_{\mathrm{HL3}} + \boldsymbol{f}_{\mathrm{HL4}} + \boldsymbol{f}_{\mathrm{HL5}} = \boldsymbol{f}_{\mathrm{HCL}} \tag{19}$$

The story for the upper chopstick is similar. The forces $\boldsymbol{f}_{\mathrm{HU1}}$ and $\boldsymbol{f}_{\mathrm{HU2}}$, formed by the more pronounced wedge between first and second fingers, serves to firmly hold and direct the middle of the chopstick in order to open and close the chopsticks for grasping food. Forces $\boldsymbol{f}_{\mathrm{HU4}}$ and $\boldsymbol{f}_{\mathrm{HU5}}$, exerted by the dip between the palm knuckle and first knuckle of the index finger, support the back of the chopstick. And vertical movement is restrained by $\boldsymbol{f}_{\mathrm{HU3}}$, once more indicated with a blob at its tail in Fig. 2, exerted by the upper part of the thumb. Again, if the chopstick is just being held steady, then these forces sum to zero. However, if food is being grasped, then they sum to $\boldsymbol{f}_{\mathrm{HCU}}$:

$$\boldsymbol{f}_{\mathrm{HU1}} + \boldsymbol{f}_{\mathrm{HU2}} + \boldsymbol{f}_{\mathrm{HU3}} + \boldsymbol{f}_{\mathrm{HU4}} + \boldsymbol{f}_{\mathrm{HU5}} = \boldsymbol{f}_{\mathrm{HCU}} \tag{20}$$

With these observations, we can decompose the GRASPFOOD$_C$ function into its plant and controller subsystems, rules CHOPSTICK$_P$ and HAND$_C$.

In those rules, we have singled out $\boldsymbol{f}_{\mathrm{CU}}$ and $\boldsymbol{f}_{\mathrm{CL}}$ as output parameters in the signature of HAND$_C$ for emphasis. They are quantities derived from the underlying hand forces, which the chopsticks react to by setting their forces appropriately. The equalities $\boldsymbol{f}_{\mathrm{HCU}} = \boldsymbol{f}_{\mathrm{CU}}$ and $\boldsymbol{f}_{\mathrm{HCL}} = \boldsymbol{f}_{\mathrm{CL}}$ again become part of the domain theory of statics.

CHOPSTICK$_P$ = (21)
choose $\boldsymbol{f}'_{\mathrm{HCU}}, \boldsymbol{f}'_{\mathrm{HCL}}$
  with $\boldsymbol{f}'_{\mathrm{HCU}} + \boldsymbol{f}'_{\mathrm{HCL}} = \mathbf{0}$
  do $\boldsymbol{f}_{\mathrm{HCU}} := \boldsymbol{f}'_{\mathrm{HCU}}, \boldsymbol{f}_{\mathrm{HCL}} := \boldsymbol{f}'_{\mathrm{HCL}}$

HAND$_C$(**out** $\boldsymbol{f}_{\mathrm{CU}}, \boldsymbol{f}_{\mathrm{CL}}$) = (22)
choose $\boldsymbol{f}'_{\mathrm{HU1}}, \boldsymbol{f}'_{\mathrm{HU2}}, \boldsymbol{f}'_{\mathrm{HU3}}, \boldsymbol{f}'_{\mathrm{HU4}}, \boldsymbol{f}'_{\mathrm{HU5}}, \boldsymbol{f}'_{\mathrm{HL1}}, \boldsymbol{f}'_{\mathrm{HL2}}, \boldsymbol{f}'_{\mathrm{HL3}}, \boldsymbol{f}'_{\mathrm{HL4}}, \boldsymbol{f}'_{\mathrm{HL5}}$
  with $\boldsymbol{f}'_{\mathrm{HU1}} + \boldsymbol{f}'_{\mathrm{HU2}} + \boldsymbol{f}'_{\mathrm{HU3}} + \boldsymbol{f}'_{\mathrm{HU4}} + \boldsymbol{f}'_{\mathrm{HU5}} +$
    $\boldsymbol{f}'_{\mathrm{HL1}} + \boldsymbol{f}'_{\mathrm{HL2}} + \boldsymbol{f}'_{\mathrm{HL3}} + \boldsymbol{f}'_{\mathrm{HL4}} + \boldsymbol{f}'_{\mathrm{HL5}} = \mathbf{0}$
    $|\boldsymbol{f}'_{\mathrm{HU1}} + \boldsymbol{f}'_{\mathrm{HU2}} + \boldsymbol{f}'_{\mathrm{HU3}} + \boldsymbol{f}'_{\mathrm{HU4}} + \boldsymbol{f}'_{\mathrm{HU5}}| =$
    $|\boldsymbol{f}'_{\mathrm{HL1}} + \boldsymbol{f}'_{\mathrm{HL2}} + \boldsymbol{f}'_{\mathrm{HL3}} + \boldsymbol{f}'_{\mathrm{HL4}} + \boldsymbol{f}'_{\mathrm{HL5}}| \geq D$
  do $\boldsymbol{f}_{\mathrm{HU1}} := \boldsymbol{f}'_{\mathrm{HU1}}, \boldsymbol{f}_{\mathrm{HU2}} := \boldsymbol{f}'_{\mathrm{HU2}}, \boldsymbol{f}_{\mathrm{HU3}} := \boldsymbol{f}'_{\mathrm{HU3}}, \boldsymbol{f}_{\mathrm{HU4}} := \boldsymbol{f}'_{\mathrm{HU4}}, \boldsymbol{f}_{\mathrm{HU5}} := \boldsymbol{f}'_{\mathrm{HU5}},$
    $\boldsymbol{f}_{\mathrm{HL1}} := \boldsymbol{f}'_{\mathrm{HL1}}, \boldsymbol{f}_{\mathrm{HL2}} := \boldsymbol{f}'_{\mathrm{HL2}}, \boldsymbol{f}_{\mathrm{HL3}} := \boldsymbol{f}'_{\mathrm{HL3}}, \boldsymbol{f}_{\mathrm{HL4}} := \boldsymbol{f}'_{\mathrm{HL4}}, \boldsymbol{f}_{\mathrm{HL5}} := \boldsymbol{f}'_{\mathrm{HL5}},$
    $\boldsymbol{f}_{\mathrm{CU}} := \boldsymbol{f}'_{\mathrm{HU1}} + \boldsymbol{f}'_{\mathrm{HU2}} + \boldsymbol{f}'_{\mathrm{HU3}} + \boldsymbol{f}'_{\mathrm{HU4}} + \boldsymbol{f}'_{\mathrm{HU5}},$
    $\boldsymbol{f}_{\mathrm{CL}} := \boldsymbol{f}'_{\mathrm{HL1}} + \boldsymbol{f}'_{\mathrm{HL2}} + \boldsymbol{f}'_{\mathrm{HL3}} + \boldsymbol{f}'_{\mathrm{HL4}} + \boldsymbol{f}'_{\mathrm{HL5}},$
    grasped := TRUE

The above completes our brief treatment of chopstick use via statics.

[6] N.B. In reality, many guides to eating with chopsticks recommend all sorts of alternative configurations for holding chopsticks — the reader may get a good idea of the various possibilities from a simple Google search on the topic of eating with chopsticks. The configuration described here is the only one that the first author has found that permits both adequate chopstick maneuvreability and sufficient deployable resultant force, especially when it comes to bigger pieces of food. A lot depends on the precise development of the musculature in the forearm of an individual, and how it is able to compel the fingers to exert forces in specific directions. Starting young helps a lot.

7. Continuous Controller Synthesis

The reader may well have noticed that there are some slightly unnatural aspects of the account of chopstick use that we gave. The ASM rules in the preceding section were the usual kind of discrete ASM rules. However, grasping via chopsticks is not the usual kind of discrete event control system. In particular, in line with the behaviour of all physical systems, both the chopsticks and the food react instantaneously to the force exerted by the other, and not to some previous value maintained by the other, the latter being what one would expect in a normal discrete event control system. We handled this via the domain theory, which demanded that the opposed forces exactly matched, without giving any inkling as to how this might be accomplished.

In a more realistic account, the force applied by the chopsticks to the food moves smoothly from zero to a value sufficient to ensure grasping, and the food senses this and smoothly reacts by offering a matching resistive force. The sudden assignment to equal and opposite values in the discrete picture is replaced by a pair of differential equations which state that the derivatives of the chopstick and food forces are equal and opposite over time, which together with initial conditions stating that both are zero, guarantees that the forces themselves remain equal and opposite. Obviously this is again an oversimplification of reality, but it is sufficient to illustrate the next chapter of our controller synthesis story.

Incorporating these insights into the ASM framework requires an extension of ASM to include continuously varying behaviours as well as discrete changes. In [3], a work subsequently expanded and elaborated in [5] and [4], the authors give such an extension which we briefly recapitulate now.

7.1. Continuous ASM

We partition the variables into two subsets: the **mode variables**, whose types are discrete sets, and the **pliant variables**, whose types include topologically dense sets, and which are permitted to evolve both continuously and via discrete changes. By restricting to mode variables alone, we recover the conventional discrete ASM framework.

Time is modelled as an interval $\mathcal{T}$ of the real numbers $\mathbb{R}$, with a finite left endpoint for the initial state, and with a right endpoint which is finite or infinite, as needed. $\mathcal{T}$ partitions into a sequence of left-closed right-open intervals, $\langle [t_0 \ldots t_1), [t_1 \ldots t_2), \ldots \rangle$, the coarsest partition such that all discontinuous changes take place at some boundary point t_i. (The actual time points t_i are determined by the runtime behaviour of the system during some run, as becomes clear from the account below.) Mode variables are constant on each of these intervals, while pliant variables evolve continuously. Otherwise arbitrary continuous evolution is constrained within reasonable bounds by three main restrictions:

I Zeno: there is a constant δ_{Zeno}, such that for all i needed, $t_{i+1} - t_i \geq \delta_{\mathsf{Zeno}}$.

II Limits: for every variable x, for every time $t \in \mathcal{T}$, and with $\delta > 0$, the left limit $\lim_{\delta \to 0} x(t-\delta)$ written $\overrightarrow{x(t)}$ and right limit $\lim_{\delta \to 0} x(t+\delta)$, written $\overleftarrow{x(t)}$ exist, and for every t, $x(t) = \overleftarrow{x(t)}$.

III Differentiability: The behaviour of every pliant variable x in the interval $[t_i \ldots t_{i+1})$ is given by the solution of a well posed initial value problem $\mathcal{D}\,xs = \phi(xs, t)$ (where xs is an appropriate vector of pliant variables, and $\mathcal{D}$ is the time derivative).

Regarding these three conditions, we make the following additional comments. Re. I, the presence or absence of Zeno behaviour is most often a property of the global reachability relation of a system permitting the kind of hybrid behaviour we are admitting, often depending sensitively on the relative values of various constants in the system model. So I is more of a desirable goal and *aide memoire* than a condition that could be imposed as a static restriction. Re II, the admittance of different left and right limits is just what is needed to accommodate discontinuous changes that take place instantaneously. The

space of functions defined by II is well studied in stochastic analysis, where it goes by the name of càdlàg. Re III, as well as differential equations *per se*, we admit direct assignments of continuous behaviour, and even implicit assignment to any function obeying a stated set of restrictions, provided *these are capable of being defined by a well posed initial value problem*. In practice, this means restriction to absolutely continuous functions; they have the property of being solutions to well posed initial value problems in the sense of Carathéodory, see [10].

The two kinds of variable (mode and pliant) are reflected in two kinds of transitions: mode and pliant. Mode transitions, given by rules of the form (23) below, just record discrete transitions from before-values to after-values of variables, with the use of the left limit for before-values and right limit for after-values giving an instantaneous interpretation to the semantics of these transitions. Both kinds of variable can be subject to a mode transition, and in (23), which is a typical ASM rule syntax for an instantaneous transition, in which the variables have been decorated with the relevant limit information,[7] we single out inputs *is* and outputs *os* in the signature of OP.

$$\begin{array}{l} \textsc{Op}(\textbf{in } \overrightarrow{is}, \textbf{out } \overleftarrow{os}) \ = \\ \textbf{if } \ guard(\overleftarrow{xs}, \overrightarrow{is}) \ \textbf{ then choose } \ \overleftarrow{xs}, \overleftarrow{os} \ \textbf{ with } \ rel(\overleftarrow{xs}, \overrightarrow{xs}, \overrightarrow{is}, \overleftarrow{os}) \\ \quad \textbf{do } \ xs, os \ := \ \overleftarrow{xs}, \overleftarrow{os} \end{array} \tag{23}$$

Pliant transitions describe continuous changes for pliant variables. While a mode transition captures a single before-/after-value pair, a pliant transition is a family of before-/after-value pairs parameterized by the relevant time interval $[t_i \ldots t_{i+1})$. The before-value is, in each case, the value at t_i, while the after-value refers to an arbitrary time in the interval, so the two values are temporally separated. A rule for a pliant transition can be written as in (24), where the symbol $\stackrel{c}{=}$ syntactically distinguishes a pliant transition from a mode transition.

$$\begin{array}{l} \textsc{PliOp}(\textbf{in } is(t \in (t_{\mathrm{L}(t)} \ldots t_{\mathrm{R}(t)})), \textbf{out } os(t \in (t_{\mathrm{L}(t)} \ldots t_{\mathrm{R}(t)}))) \ \stackrel{c}{=} \\ \textbf{if } \ IV(xs(t_{\mathrm{L}(t)})) \ \textbf{ and } \ guard(xs(t_{\mathrm{L}(t)})) \ \textbf{ then with } \ rel(xs, is, os, t) \\ \quad \textbf{do } \ xs(t), os(t) \ := \ \textbf{solve } DE(xs(t), is(t), os(t), t) \end{array} \tag{24}$$

In (24), $\mathrm{L}(t) = max\{i \mid t_i \leq t\}$ and $\mathrm{R}(t) = min\{i \mid t_i > t\}$ so that we do not have to statically know the index i for the interval $[t_i \ldots t_{i+1})$, thus making the notation generic. Furthermore, *IV* and *guard* refer to the initial value and any additional guard restriction that apply for the initial value problem in $[t_i \ldots t_{i+1})$.[8] The **solve** keyword announces that what follows is a differential equation *DE* that defines the needed behaviour of the initial value problem, while *rel* expresses any additional constraints that must hold beyond *DE*. Inputs *is* and outputs *os* (shown as depending on the whole interval $(t_{\mathrm{L}(t)} \ldots t_{\mathrm{R}(t)})$) again appear in the signature. If, as can often happen, we know the form of the continuous behaviour that we want (in contrast to merely knowing a differential equation that specifies it), then we can replace the **solve** clause with a straightforward assignment using a **do**, bearing in mind that what is being expressed is a time-indexed family of individual assignments.

We say that a continuous ASM ruleset is **well formed** iff the initial state is regarded as being established by the (after-state of a true-guarded) initial mode transition,[9] and:

- Every enabled mode transition is feasible, i.e. has an after-state, and on its completion enables a pliant transition (but does not enable any mode transition). (25)

[7] So the overarrows are just semantic decoration, and not part of the syntax.

[8] In a pliant transition, it is often convenient to separate guard restrictions applying to mode variables, which are bound to remain true during the whole of the ensuing pliant transition, from initial value constraints on pliant variables, which are prone to failure immediately that the continuous evolution commences. We can put the former in *guard* and the latter in *IV*.

[9] Formulating the initial state this way simplifies the description of the formal operational semantics in [5] a little.

- Every enabled pliant transition is feasible, i.e. has a time-indexed family of after-states, and EITHER: (26)
  - (i) During the run of the pliant transition a mode transition becomes enabled. It preempts the pliant transition, defining its end. ORELSE
  - (ii) During the run of the pliant transition it becomes infeasible: finite termination. ORELSE
  - (iii) The pliant transition continues indefinitely: nontermination.

Although it is rather selfevident that the preceding informal account of the continuous extension of ASM misses out much of the fine detail of a complete operational semantics, in the kind of relatively simple continuous behaviours that are usually of interest in applications like ours, intuition is a powerful guide to the required behaviour, and the full details may be safely relegated to a more precise treatment, as can be found in [5]. Accordingly, in this paper, we content ourselves with the sketch just given.

7.2. Continuous ASM as an Extension of Conventional ASM

Given the above outline of continuous ASM, we can regard it as a conservative extension of traditional, discrete ASM, by embedding a generic conventional discrete ASM system into continuous ASM in the following manner:

- We consider all of the original discrete ASM rules as mode rules.
- We decide on a fixed duration $\delta_t > 0$ for all the pliant transitions.
- We determine that each state of the discrete event ASM system will persist for δ_t.
- We add continuous ASM rules for all needed pliant transitions (perhaps only one), that in effect just skip in a continuous manner (by setting the time derivatives of all ASM state variables to 0 where needed).
- We add a time variable *t* say, and enable all mode transitions after the elapse of any integral multiple of δ_t (by adding an expression like "$t/\delta_t \in \mathbb{N}$" to each mode rule guard).

We see that the above merely expresses a specific instance of that which is normally assumed without comment in discrete transition systems. Thus (firstly), in a conventional discrete transition system, transitions are normally understood to take place instantaneously. Also (secondly), as soon as such a transition has taken place, the state variables already have the values that will enable the next transition (since they do not change in between transitions). However (thirdly), this next transition does not take place straight away (which would imply that it happened at the same instant as the preceding transition, and hence, by induction, that the entire run of the discrete transition system took place at a single moment of time), but is normally assumed to take place some time later.

7.3. Continuous ASM Refinement

In the context of the preceding account, a natural question arises regarding the impact on ASM refinement. In fact, given the clean way our continuous extension of ASM extends the discrete framework, and the extreme flexibility of the notion of discrete ASM refinement, very little has to change. It is possible to set up a notion of continuous ASM refinement by allowing, in a generic (m,n) diagram, the abstract and concrete operation sequences $\mathrm{XOP}(x,x')$ and $\mathrm{YOP}(y,y')$ to consist of arbitrary sequences of interleaved mode and pliant transitions, rather than them being sequences of exclusively mode transitions as in the discrete case. The POs (6) and (7) remain unchanged.

In this case, XOP defines a function from time to state values, lasting from the beginning of the first operation of XOP to the end of the last operation of XOP, taking suitable limits if these are needed to

obtain precise values. Similarly for YOP. These initial and final value pairs are the ones appearing in the equivalences $R(x,y)$ and $R(x',y')$ in the POs. And with these technical details understood, the principle of abutting occurrences of (m,n) diagrams to form a simulation between runs remains unchanged.

It is worth pointing out that the above remarks, in not demanding more than the original POs of the discrete version, do *not* stipulate any particular restriction on the relative passage of time in the abstract and concrete models, nor on any other aspect not explicitly mentioned. We can interpret the great flexibility of the original ASM refinement notion as encouraging the greatest possible flexibility in the continuous extension, accompanied of course, by the responsibility of justifying any particular decision taken in any particular application, against the requirements.

7.4. Continuous Controller Synthesis

We can ask how the process of separating a set of rules into controller and plant rules works, when we have pliant as well as mode transitions. In fact, the process is very similar to what went before. Since mode rules are identical to the rules we considered earlier, there is nothing new for them. For pliant rules, they also have a *guard* and a *rel*, and for these we demand the same conditions as previously. But there is also the **solve** clause. We need to stipulate that it separates cleanly into controller and plant in the same way that *guard* and *rel* do so that the rule as a whole splits neatly.

The tuple of differential equations $\mathcal{D}\,xs = \phi(xs,t)$ contained in the **solve** clause of a *Sys* rule naturally splits into two: $\mathcal{D}\,xs_C = \phi_C(xs,t)$ and $\mathcal{D}\,xs_P = \phi_P(xs,t)$. But there is no *a priori* guarantee that $\phi_C(xs,t)$ contains only the variables xs_C, xs_P^c, and $\phi_P(xs,t)$ contains only the variables xs_P, xs_C^p. So for admissibility, we must additionally demand the following: $\phi_C(xs,t)$ contains occurrences of only the variables xs_C, xs_P^c, and $\phi_P(xs,t)$ contains occurrences of only the variables xs_P, xs_C^p.

With these provisos, the pliant counterparts of (1) and (2)-(3) become, respectively, (24) and:

$$\begin{array}{l}\textsc{PliOp}_C(\textbf{in}\ is_C(t \in (t_{\mathrm{L}(t)} \ldots t_{\mathrm{R}(t)})), \textbf{out}\ os_C(t \in (t_{\mathrm{L}(t)} \ldots t_{\mathrm{R}(t)}))) \stackrel{\mathrm{c}}{=} \\ \textbf{if}\ \ IV_C(xs_C(t_{\mathrm{L}(t)}), xs_P^c(t_{\mathrm{L}(t)}))\ \ \textbf{and}\ \ guard_C(xs_C(t_{\mathrm{L}(t)}), xs_P^c(t_{\mathrm{L}(t)})) \\ \textbf{then with}\ \ rel_C(xs_C, xs_P^c, is_C, os_C, t) \\ \quad \textbf{do}\ \ xs_C(t), os_C(t)\ :=\ \textbf{solve}\ DE_C(xs_C(t), xs_P^c(t), is_C(t), os_C(t), t)\end{array} \tag{27}$$

$$\begin{array}{l}\textsc{PliOp}_P(\textbf{in}\ is_P(t \in (t_{\mathrm{L}(t)} \ldots t_{\mathrm{R}(t)})), \textbf{out}\ os_P(t \in (t_{\mathrm{L}(t)} \ldots t_{\mathrm{R}(t)}))) \stackrel{\mathrm{c}}{=} \\ \textbf{if}\ \ IV_P(xs_P(t_{\mathrm{L}(t)}), xs_C^p(t_{\mathrm{L}(t)}))\ \ \textbf{and}\ \ guard_P(xs_P(t_{\mathrm{L}(t)}), xs_C^p(t_{\mathrm{L}(t)})) \\ \textbf{then with}\ \ rel_P(xs_P, xs_C^p, is_P, os_P, t) \\ \quad \textbf{do}\ \ xs_P(t), os_P(t)\ :=\ \textbf{solve}\ DE_P(xs_P(t), xs_C^p(t), is_P(t), os_P(t), t)\end{array} \tag{28}$$

It is now clear that the embedding of discrete ASMs into continuous ASMs outlined at the end of the last section is admissible in the extended sense just discussed, provided the original discrete ASM system is admissible, so that the properties derived for controller synthesis in Sections 2 and 3 carry through essentially unchanged.

We can also say that the remarks about domain theories made in Section 4 remain true in the continuous context, particularly when we recognise that differential equations couple instantaneous rates of change of variables at some time t, to values of variables at *the same* time t, in a manner analogous to the situation we have already seen in the statics treatment of chopstick use in Section 6. And if we further observe that $(1,1)$ refinement for an abstract/concrete pair of pliant transitions in the continuous case can be regarded as a time-parameterised family of discrete $(1,1)$ refinement relations, relating abstract/concrete before-values at $t_{\mathrm{L}(t)}$ to abstract/concrete after-values at t (for all $t \in (t_{\mathrm{L}(t)} \ldots t_{\mathrm{R}(t)})$), then the results on complete refinement in Section 5 carry through in the appropriate manner as well.

8. Continuous Grasping

We now revisit the chopsticks case study from Section 6 in the continuous ASM framework, to see how the latter can lend it a more persuasive air.

As before, to keep things relatively simple, we restrict the modeling to that of forces only (albeit now allowing them to vary continuously), neglecting other issues as in Section 6. This avoids complications arising from having to consider movement of either the food or the chopsticks, or distortions of the shape of either the food or chopsticks consequent on them experiencing the forces that we model, and keeps the model that we present within a relatively limited space.

We concentrate on elaborating the simpler model in Section 6.1. Time $t = 0$ triggers the initial mode rule:

$$\begin{array}{l}\textsc{start} = \\ \textbf{if } t = 0 \textbf{ then} \\ \quad \textbf{do } \mathit{mode} := \mathit{grasping}, \mathit{grasped} := \mathit{undef}, \\ \qquad \boldsymbol{f}_{\mathrm{FU}} := \mathbf{0}, \boldsymbol{f}_{\mathrm{FL}} := \mathbf{0}, \boldsymbol{f}_{\mathrm{HCU}} := \mathbf{0}, \boldsymbol{f}_{\mathrm{HCL}} := \mathbf{0}\end{array} \tag{29}$$

The *grasping* mode enables the following pliant rule:

$$\begin{array}{l}\textsc{grasping} \stackrel{c}{=} \\ \textbf{if } \mathit{mode} = \mathit{grasping} \textbf{ then} \\ \quad \textbf{do } \boldsymbol{f}_{\mathrm{FU}}, \boldsymbol{f}_{\mathrm{FL}}, \boldsymbol{f}_{\mathrm{HCU}}, \boldsymbol{f}_{\mathrm{HCL}} := \\ \qquad \textbf{solve } [\, \mathcal{D}\boldsymbol{f}_{\mathrm{FU}}, \mathcal{D}\boldsymbol{f}_{\mathrm{FL}}, \mathcal{D}\boldsymbol{f}_{\mathrm{HCU}}, \mathcal{D}\boldsymbol{f}_{\mathrm{HCL}} \,] = [\, \mathbf{e}_{\mathrm{z}}, -\mathbf{e}_{\mathrm{z}}, -\mathbf{e}_{\mathrm{z}}, \mathbf{e}_{\mathrm{z}} \,]\end{array} \tag{30}$$

This rule causes the forces $\boldsymbol{f}_{\mathrm{FU}}$, $\boldsymbol{f}_{\mathrm{FL}}$, $\boldsymbol{f}_{\mathrm{HCU}}$, $\boldsymbol{f}_{\mathrm{HCL}}$ to acquire suitable pairwise equal and opposite rates of change, of magnitude 1, oriented along the unit vector of the z axis. This causes these forces to change continuously (although in fact non-smoothly[10]) away from zero at a uniform rate. The continuous grasping persists until a time t_{STOP}, when it is determined whether enough force has been applied to hold the food:

$$\begin{array}{l}\textsc{stopgrasped} = \\ \textbf{if } t = t_{\mathrm{STOP}} \wedge \boldsymbol{f}_{\mathrm{HCU}} \geq D \textbf{ then} \\ \quad \textbf{do } \mathit{mode} := \mathit{stop}, \mathit{grasped} := \textsc{true}\end{array} \tag{31}$$

$$\begin{array}{l}\textsc{stopdislodged} = \\ \textbf{if } t = t_{\mathrm{STOP}} \wedge \boldsymbol{f}_{\mathrm{HCU}} < D \textbf{ then} \\ \quad \textbf{do } \mathit{mode} := \mathit{stop}, \mathit{grasped} := \textsc{false}\end{array} \tag{32}$$

The stopped mode just enters a pliant final state:

$$\textsc{f-idle} \stackrel{c}{=} \textbf{if } \mathit{mode} = \mathit{stop} \textbf{ then do } \mathsf{skip} \tag{33}$$

The above is all consistent with the domain theory (16), although the variables in the theory would have to be interpreted as functions of time, and the theory itself would have to be augmented by various facts concerning time and the additional variables introduced above, in order that the natural continuous counterparts of the statements in (5) could hold.[11]

[10] Since the derivatives of the forces jump discontinuously at $t = 0$, the forces themselves, though continuous, experience a kink at $t = 0$.

[11] The domain theory would also have to be supplemented with a background theory of facts about calculus, continuous mathematics etc., as needed.

8.1. Decomposing Continuous Grasping

We now look at applying the decomposition strategy discussed earlier to the above integrated model. We assume that the chopsticks, as controller, are in charge, and own variables like *mode* and *grasped*. We decompose the rules above one by one, starting with START:

$$\begin{array}{l}\textsc{Start}_C\ = \\ \textbf{if}\ \ t = 0\ \ \textbf{then} \\ \quad \textbf{do}\ \ \mathit{mode} := \mathit{grasping},\ \mathit{grasped} := \mathit{undef}, \\ \qquad \boldsymbol{f}_{\mathrm{HCU}} := \mathbf{0},\ \boldsymbol{f}_{\mathrm{HCL}} := \mathbf{0}\end{array} \tag{34}$$

$$\begin{array}{l}\textsc{Start}_P\ = \\ \textbf{if}\ \ t = 0\ \ \textbf{then} \\ \quad \textbf{do}\ \ \boldsymbol{f}_{\mathrm{FU}} := \mathbf{0},\ \boldsymbol{f}_{\mathrm{FL}} := \mathbf{0}\end{array} \tag{35}$$

Next, the decomposition of the GRASPING rule. This yields:

$$\begin{array}{l}\textsc{Grasping}_C(\textbf{out}\ \boldsymbol{of}_{\mathrm{HCU}}, \boldsymbol{of}_{\mathrm{HCL}})\ \stackrel{\mathrm{c}}{=} \\ \textbf{if}\ \ \mathit{mode} = \mathit{grasping}\ \ \textbf{then} \\ \quad \textbf{do}\ \ \boldsymbol{f}_{\mathrm{HCU}},\ \boldsymbol{f}_{\mathrm{HCL}} := \textbf{solve}\ [\ \mathcal{D}\boldsymbol{f}_{\mathrm{HCU}}, \mathcal{D}\boldsymbol{f}_{\mathrm{HCL}}\ ] = [\ -\mathbf{e}_z, \mathbf{e}_z\ ], \\ \qquad \boldsymbol{of}_{\mathrm{HCU}} := \boldsymbol{f}_{\mathrm{HCU}},\ \boldsymbol{of}_{\mathrm{HCL}} := \boldsymbol{f}_{\mathrm{HCL}}\end{array} \tag{36}$$

$$\begin{array}{l}\textsc{Grasping}_P(\textbf{in}\ \boldsymbol{if}_{\mathrm{HCU}}, \boldsymbol{if}_{\mathrm{HCL}})\ \stackrel{\mathrm{c}}{=} \\ \textbf{if}\ \ \mathit{mode} = \mathit{grasping}\ \ \textbf{then} \\ \quad \textbf{do}\ \ \boldsymbol{f}_{\mathrm{FU}} := -\boldsymbol{if}_{\mathrm{HCU}},\ \boldsymbol{f}_{\mathrm{FL}} := -\boldsymbol{if}_{\mathrm{HCL}}\end{array} \tag{37}$$

The above rules display a slightly more complex manner of decomposition than we have considered hitherto. Instead of merely partitioning the variables and determining that subsystem B has read access to some of the variables owned by subsystem A, we have introduced input and output variables that do this job explicitly. So the chopsticks have output variables $\boldsymbol{of}_{\mathrm{HCU}}$ and $\boldsymbol{of}_{\mathrm{HCL}}$, which are just copies of variables $\boldsymbol{f}_{\mathrm{HCU}}$ and $\boldsymbol{f}_{\mathrm{HCL}}$, and the food has input variables $\boldsymbol{if}_{\mathrm{HCU}}$ and $\boldsymbol{if}_{\mathrm{HCL}}$, which are used to read the relevant values in. Thus, the modeling is a now little different in that the food explicitly reacts to the forces it senses, by generating equal and opposite forces of its own, instead of generating these forces directly as a result of solving separate differential equations for these forces, as in (30). Technically, we have substituted equals for equals, but have gone beyond the simple syntactic transformation described earlier in the paper. It is a natural temptation to do this at the more realistic and practical level of modeling that we have reached. Since the new variables are just copies of existing ones, only trivial modifications are needed to the earlier formal results, and it would merely add clutter to complicate the earlier theory by including them.

Next are the STOP rules:

$$\begin{array}{l}\textsc{StopGrasped}_C\ =\ \textbf{if}\ \ t = t_{\mathrm{STOP}}\ \wedge\ \boldsymbol{f}_{\mathrm{HCU}} \geq D\ \ \textbf{then} \\ \quad \textbf{do}\ \ \mathit{mode} := \mathit{stop}, \mathit{grasped} := \textsc{true}\end{array} \tag{38}$$

$$\begin{array}{l}\textsc{StopDislodged}_C\ =\ \textbf{if}\ \ t = t_{\mathrm{STOP}}\ \wedge\ \boldsymbol{f}_{\mathrm{HCU}} < D\ \ \textbf{then} \\ \quad \textbf{do}\ \ \mathit{mode} := \mathit{stop}, \mathit{grasped} := \textsc{false}\end{array} \tag{39}$$

$$\textsc{StopGrasped}_P\ =\ \textbf{if}\ \ t = t_{\mathrm{STOP}}\ \ \textbf{then do}\ \ \mathsf{skip} \tag{40}$$

$$\textsc{StopDislodged}_P\ =\ \textbf{if}\ \ t = t_{\mathrm{STOP}}\ \ \textbf{then do}\ \ \mathsf{skip} \tag{41}$$

And lastly the final idle rules:

$$\text{F-IDLE}_C \stackrel{c}{=} \textbf{if } \mathit{mode} = \mathit{stop} \textbf{ then do } \mathsf{skip} \tag{42}$$

$$\text{F-IDLE}_P \stackrel{c}{=} \textbf{if } \mathit{mode} = \mathit{stop} \textbf{ then do } \mathsf{skip} \tag{43}$$

The preceding shows that the controller synthesis procedure that we have described is as applicable to the continuous extension of ASM as it is to the discrete version. We could now go on to apply the same approach to create a continuous version of the decomposed hand+chopsticks model, but although there is no technical impediment to doing so, it would introduce a lot of complexity into the model description, without bringing any genuinely new insight to the table. For this reason, we do not pursue it in detail.

9. Unsynchronised Continuous Controller Synthesis

In this section we explore a phenomenon connected with controller synthesis that is exclusive to the continuous case. The situation arises as follows. Suppose that we have a physical process θ that is to be controlled using some input signal u, using a first order DE:

$$\mathcal{D}\theta = u \tag{44}$$

For our purposes, it will not matter exactly what θ or u might actually be. More important is the fact that, in line with the overwhelming majority of control applications today, we regard u as being set by a digital process, which instructs actuators to hold particular values on θ's input signal until re-instructed to hold new ones by a fresh digital command. So u is in fact a piecewise constant signal, and at the level of abstraction of interest to us, we consider the updates to u to be performed by mode events. In a unified modeling framework, assuming a single period of behaviour lasting (perhaps) 10 time units, we can describe the scenario mentioned in the ruleset below. In those rules, we assume that at integral times $t \in \mathbb{N}$, the new value of the control signal is chosen (arbitrarily, for simplicity, from U, the type of u), and that in between these integral times, u is held constant and θ obeys (44):

$$\begin{array}{l}\text{START} = \\ \textbf{if } t = 0 \textbf{ then} \\ \quad \textbf{do } \mathit{mode} := \mathit{behave}, \\ \qquad \theta := \theta_0,\ u := u_0\end{array} \tag{45}$$

$$\begin{array}{l}\text{BEHAVE} \stackrel{c}{=} \\ \textbf{if } \mathit{mode} = \mathit{behave} \textbf{ then} \\ \quad \textbf{do } \theta, u := \textbf{solve } [\ \mathcal{D}\theta, \mathcal{D}u\ ] = [\ u, 0\ ]\end{array} \tag{46}$$

$$\begin{array}{l}\text{UPDATEU} = \\ \textbf{if } \mathit{mode} = \mathit{behave} \wedge t \in \mathbb{N} \wedge t \leq 9 \textbf{ then choose } u' \in U \\ \quad \textbf{do } u := u'\end{array} \tag{47}$$

$$\begin{array}{l}\text{STOPU} = \\ \textbf{if } \mathit{mode} = \mathit{behave} \wedge t \in \mathbb{N} \wedge t = 10 \textbf{ then} \\ \quad \textbf{do } \mathit{mode} := \mathit{stop},\ u := 0\end{array} \tag{48}$$

$$\begin{array}{l}\text{IDLEU} \stackrel{c}{=} \\ \textbf{if } \mathit{mode} = \mathit{stop} \textbf{ then do } \mathsf{skip}\end{array} \tag{49}$$

Now we can examine what happens when we decompose the above system according to our controller synthesis strategy described earlier. In contrast to our earlier practice, we list the controller rules first, plant rules afterwards, rather than interleaving them. Here are the controller rules.

$$\begin{array}{l}\textsc{start}_C\ = \\ \textbf{if}\ t=0\ \textbf{then} \\ \quad \textbf{do}\ \mathit{mode} := \mathit{behave}, \\ \qquad u := u_0\end{array} \tag{50}$$

$$\begin{array}{l}\textsc{behave}_C \stackrel{c}{=} \\ \textbf{if}\ \mathit{mode}=\mathit{behave}\ \textbf{then} \\ \quad \textbf{do}\ u := \textbf{solve}\ \mathcal{D}u=0\end{array} \tag{51}$$

$$\begin{array}{l}\textsc{updateU}_C\ = \\ \textbf{if}\ \mathit{mode}=\mathit{behave}\ \wedge\ t\in\mathbb{N}\ \wedge\ t\leq 9\ \textbf{then choose}\ u'\in U \\ \quad \textbf{do}\ u := u'\end{array} \tag{52}$$

$$\begin{array}{l}\textsc{stopU}_C\ = \\ \textbf{if}\ \mathit{mode}=\mathit{behave}\ \wedge\ t\in\mathbb{N}\ \wedge\ t=10\ \textbf{then} \\ \quad \textbf{do}\ \mathit{mode} := \mathit{stop}, \\ \qquad u := 0\end{array} \tag{53}$$

$$\begin{array}{l}\textsc{idleU}_C \stackrel{c}{=} \\ \textbf{if}\ \mathit{mode}=\mathit{stop}\ \textbf{then do}\ \mathsf{skip}\end{array} \tag{54}$$

And now the plant rules.

$$\begin{array}{l}\textsc{start}_P\ = \\ \textbf{if}\ t=0\ \textbf{then do}\ \theta := \theta_0\end{array} \tag{55}$$

$$\begin{array}{l}\textsc{behave}_P \stackrel{c}{=} \\ \textbf{if}\ \mathit{mode}=\mathit{behave}\ \textbf{then do}\ \theta := \textbf{solve}\ \mathcal{D}\theta=u\end{array} \tag{56}$$

$$\begin{array}{l}\textsc{updateU}_P\ = \\ \textbf{if}\ \mathit{mode}=\mathit{behave}\ \wedge\ t\in\mathbb{N}\ \wedge\ t\leq 9\ \textbf{then do}\ \mathsf{skip}\end{array} \tag{57}$$

$$\begin{array}{l}\textsc{stopU}_P\ = \\ \textbf{if}\ \mathit{mode}=\mathit{behave}\ \wedge\ t\in\mathbb{N}\ \wedge\ t=10\ \textbf{then do}\ \mathsf{skip}\end{array} \tag{58}$$

$$\begin{array}{l}\textsc{idleU}_P \stackrel{c}{=} \\ \textbf{if}\ \mathit{mode}=\mathit{stop}\ \textbf{then do}\ \mathsf{skip}\end{array} \tag{59}$$

What is interesting about the decomposed version of the rules, is rules (57) and (58). These are mode rules that skip. Our decomposition technique has generated mode rules that do nothing. Mode rules that do nothing are unlike pliant rules that do nothing. Let us pause momentarily to examine why.

Since real time is a first class citizen in the continuous ASM framework, and behaviour is continuous, a continuous ASM system model must always be obeying *some* rule. Given the finite nature of continuous ASM system descriptions, the rule that the system will be obeying will, almost always,[12] be a pliant rule. Thus, even if the system is to remain in some particular fixed state over a period of time, this idle behaviour must be explicitly specified somehow.

[12] 'Amost always' is intended in the technical, measure theoretic sense of 'almost everywhere, in the set of times'.

The case for mode rules that skip is different. The runtime behaviour of a mode rule has no duration. If the rule effects no change in any variable, then one would normally expect the continuous behaviour of the variables that was being implemented by the pliant rule(s) that was(were) active immediately prior to the mode rule's skip to continue.

We said 'would normally expect ... to continue' rather than 'will continue' just now, for the following continuous-ASM-specific reason. Let us say for the sake of argument that a mode rule that skips, *MoSkip*, is the only rule schedulable at a given moment. Then executing *MoSkip* introduces a scheduling point into the dynamics. *With the mode rule MoSkip included in the run*, the previously running pliant rule(s), *PliBefore* (suppose there is just one), which was preempted by *MoSkip*, may no longer be enabled after *MoSkip* (even though *MoSkip* did nothing), since *PliBefore* may have disabled itself through its own activities, for example by altering some variables in such a way that *PliBefore*'s guards became false. And *MoSkip*, once (trivially) completed, may enable some new pliant rule(s), *PliAfter* (suppose again that there is just one), which then takes over and gives rise to new behaviour. Now suppose that *MoSkip* is *not* scheduled at that moment (for example by considering a system that does not contain *MoSkip* but is otherwise identical). *Without the mode rule MoSkip included in the run*, the previously running *PliBefore* may well continue to effect the behaviour it was previously implementing (even though it may have disabled its own guards — since it only needs to check its guards at the beginning of its execution), and since there is no mode rule occurrence to preempt *PliBefore*, the new pliant rule spoken of in the preceding case, *PliAfter*, remains locked out and cannot cause the new behaviour mentioned. Therefore, whether or not a mode skip actually executes at a given point in a system run, can make a difference.

We regard (the semantics of) each run of a well formed system as the set of time dependent functions —one for each variable of the system— that define the value of each of the system's variables at each time within the duration of the run. With this definition, we say that a set of continuous ASM rules is **normal**, iff, no two distinct pliant rules of the system can give rise to the same time dependent valuations for the set of all variables over any open interval of time, and, whenever all mode rules that merely skip are removed, then the set of runs is unchanged (i.e. no runs are added, no runs are removed, and no runs are changed in any way[13]). Note that whether or not a set of rules containing mode skips is normal or not may depend on the mode skip rules' guards. These guards may have true-sets that are sufficiently small that they prevent the rules being scheduled at times when they might otherwise cause some visible alteration in the set of runs of the system.

We can now apply these insights to our example above. We note that the guards of rules (57) and (58), which are mode skips, are strong enough to ensure they are scheduled only at the same time as nontrivial mode rules (52) and (53) respectively. Therefore, we can omit them from the aggregated set of controller and plant rules, without altering the time dependent valuation function of any variable.

Theorem 9.1. *Let Full be a system of continuous ASM rules. Let Brief be the system derived from Full by removing all the mode skip rules from Full. Suppose Full is a well formed normal system of rules. Then Brief is a complete refinement of Full.*

Proof: If *Full* has no mode skip rules, then *Full* and *Brief* are identical — the collection of $(1,1)$ diagrams expressing an identity refinement, between, on the one hand, a simultaneously enabled set of rules, and on the other hand, itself, is evidently a complete refinement, and there is nothing more to show.

[13] In other words, if a well formed system is normal, then the occurrence of a mode skip causes no observable effect, and in particular, the phenomenon of allowing *PliAfter* to run after the mode skip, as described above, certainly does not take place. In a well formed normal system, for each variable, the left limit value at the moment the mode skip occurs, is equal to the right limit value at that moment (and is, in turn, equal to the actual value at that moment), so the occurrence or not of the mode skip is not visible in the time dependent function of the values of any variable.

Otherwise, *Full* and *Brief* are different, despite which, by assumption, they have the same sets of runs (interpreted as variables' time dependent valuations). To show a complete refinement, we have to exhibit an appropriate set of (m,n) refinement diagrams. We have to supply enough of these to cater for all ways in which mode skips might (or not) have contributed to any run in particular.

The set of (m,n) diagrams consists of the following three batches. Firstly, it has all $(1,1)$ diagrams expressing an identity refinement, between, on the one hand, a simultaneously enabled set of rules containing no mode skips, and on the other hand, the same set of rules. Secondly, it contains all $(1,1)$ diagrams expressing an identity refinement, between, on the one hand, a simultaneously enabled set of rules containing at least one mode skip but also at least one non-mode skip rule, and on the other hand, the same set of rules with all the mode skips removed. For these first two batches, it is evident that the complete refinement criteria are easily met.

Thirdly, it contains all $(m,1)$ diagrams constructed according to the following criteria: (a) the sequence of m steps of *Full* is reachable, occuring in some run of *Full*; (b) the number m is odd, at least 3; (c) the first, last and all odd-indexed transitions of the m steps of *Full* are pliant transitions; (d) all even-indexed transitions of the m steps of *Full* are mode transitions defined by mode skip rules only; (e) the single step of *Brief* is a pliant transition which refines the m steps of *Full*. We must justify that the definition of the third case is both consistent, and covers all the situations needed that arise, and that are not already covered by the first two cases.

Regarding consistency, we claim that all the pliant transitions of the sequence of m steps of *Full* are defined by the same (set of simultaneously enabled) rule(s). For suppose not, and that the first transition (pliant) was defined by *Pli*1, and, following the second (mode skip) transition, the third transition (pliant) was defined by $Pli2 \neq Pli1$. Then, removing the intervening mode skip, would render *Pli*2 unschedulable in any run in which this fragment occurred, and therefore, this run (with mode skips removed) could not be a run of *Brief*. Since we assumed *Full* is well formed and normal, this is a contradiction. Since all the pliant transitions of the sequence of m steps of *Full* are defined by the same set of simultaneously enabled rules, they join together into a single behaviour, as explained in footnote 13. This single behaviour is the content of the 1 pliant transition of the *Brief* system in the $(m,1)$ diagram. The requirements of complete refinement now follow readily for this $(m,1)$ diagram.

Regarding coverage, we argue as follows. Consider a run of *Full*. Because *Full* is well formed and normal, we can remove all mode skips, to get a run of *Brief*. As argued above, when we remove the transitions consisting of mode skips alone, we are secure in the knowledge that the pliant behaviour either side can be joined into a single behaviour of *Brief*, defined by the same set of rules. Whenever this happens for k consecutive pure-mode-skip transitions, we view it as an instance of a $(2k+1,1)$ diagram of the kind constructed above. All other steps of the *Full* run and of the *Brief* run are covered by the $(1,1)$ diagrams of the first or second batch discussed above. Since *Full* is well formed and normal, there are no other runs to consider for either *Full* or *Brief*. We are done. □

9.1. skips and Multicomponent Systems

So far, we have been regarding the decomposed system of rules as a single aggregated set of rules specifying overall system behaviour, albeit that we regard one subset of the rules as belonging to the plant and the remainder as belonging to the controller. In this sense the partition of rules is a meta level concept: in the terminology of Section 2 we are talking about Sys_{C+P}, rather than the two subsystems Sys_C and Sys_P separately. In this view, we saw that we could discard the mode skip rules (57) and (58), since they are always scheduled with non-skip rules, and thus ASM scheduling semantics implied that the overall runtime semantics was unaffected by their removal. So in this view, all the mode skip rules fall into the second batch discussed in the proof of Theorem 9.1.

If we change the viewpoint, and now look at the system as two separate (sub)systems, Sys_C and Sys_P, then the picture changes. A number of questions present themselves.

The main question is: how independent are the two systems intended to be, and what are the consequences of this?

When components are designed independently of one another, they need to precisely define the mechanisms by which they interact with their companion components in the rest of the system. Usually in component based software engineering (CBSE), components have ports of some kind, through which they communicate and/or synchronise (see e.g. [1] for coverage of this large discipline). The communication/synchronisation mechanisms need to be very well understood for assemblies of such components to be able to work together at all. If we consider our subsystems Sys_C and Sys_P in this light, we see that our controller synthesis process leaves us well short by comparison with the expectations of CBSE. It simply gives us a collection of rules, partitioned according to attributed functionality. The formalised communication/synchronisation mechanisms that a CBSE view would need, are left very implicit in the structure of the rules and their interdependencies.

We can see this reality reflected in the properties of the mode skip rules (57) and (58) discussed above. Let us discuss these one by one.

The UPDATEU$_P$ rule (57) causes no plant variable update, but marks times at which the variable u may undergo discontinuous change as a result of the action of controller mode rule (52). It is thus acting as a communication mechanism to receive notice of these changes in u (even though the plant system needs to do nothing as a result of these discontinuous changes happening). If Sys_P were to be regarded as an independent component, then all aspects of this would need to be recorded in the interface that Sys_P offers to the rest of the world. Purely technically, rule UPDATEU$_P$ may be discarded without causing discomfort *provided* a particular observation holds, namely that we interpret the differential equations occurring in a continuous ASM system in the sense of Carathéodory [10]. Amongst other things, this allows the right hand sides of differential equations such as $\mathcal{D}\theta = u$ to suffer the kind of discontinuity just mentioned, without harming the existence or continuity of the resulting solutions. This is exactly the property we need to handle the effects of (52).[14]

The STOPU$_P$ rule (58) is of a different nature. It causes no plant variable update, but marks the time at which the *mode* variable changes from *behave* to *stop* by the action of the controller subsystem. As above, it is acting as a communication mechanism to receive notice of the changes in the value of *mode*. Again, if Sys_P were to be regarded as an independent component, then all aspects of this would need to be recorded in the interface that Sys_P offers to the rest of the world. However, this time, the technical optimisation that we had before is no longer available, since the plant subsystem needs to change its own behaviour as a result of the change in *mode*, by launching a different pliant rule. Before, the pliant behaviour is defined by BEHAVE$_P$, whereas after, the pliant behaviour is defined by IDLEU$_P$. If we omitted the STOPU$_P$ rule, the behaviour of the plant subsystem, *viewed in isolation*, would feature a pliant transition followed by a different pliant transition, without an intervening mode transition, violating well formedness. So we cannot omit STOPU$_P$ in the way we could before. But this should not trouble us, since STOPU$_P$ is acting as the receiver of a communication from the environment of the change in *mode*: omitting one half of a communication would be a very unusual thing to do. The fact that we might actually be able to do that in the case UPDATEU$_P$ is the more unusual possibility.

Recapitulating, viewing the Sys_C and Sys_P subsystems that our process generates as truly independent components is really carrying the separation analogy too far. Simply viewing them as two collections of rules, neglects all the interface aspects that need to be taken into account which ensure that these two systems of rules are able to cooperate fruitfully. We have seen that this turns out to hold some quite subtle connotations, since the cooperation is mediated by shared variables, which invariably leads to a

[14] As well as relating to variables that are read directly, similar remarks apply to pliant rule input variables, that happen to be governed by other system components, and which also undergo discontinuities.

very tight semantic interdependency between the two systems. This however, is the normal case for control systems, where the interplay between controller and plant is indeed predominantly via shared variables and the coupling between them is rather intimate. So the differing fortunes of the mode skip elimination strategy should not surprise us.

10. Conclusion

In this paper we have introduced the controller synthesis problem for ASM systems. The motivation was that from a goal oriented point of view, it is often more convenient to focus on overall system objectives at the outset, and to postpone detailed implementation issues, such as the specific assignment of functionality to controller or to plant, till later. This contrasts somewhat with the usual approach advocated in the ASM method, which recommends paying close attention to the classification of variables and functions into monitored and controlled categories right at the outset of system design. We believe that there is merit in both points of view, and that a goal oriented approach can be a useful adjunct to the recommended method in the early stages of design.

We showed that controller synthesis, as we have defined it, is undecidable, and we gave a safe approximation. We argued that the success of the approach invariably depends on having a suitable domain theory to bind the separate behaviours of controller and plant into a consistent whole, and we discussed the nature of such a domain theory in some detail. We also showed that our notion of controller synthesis was quite closely related to the previously existing and more general ASM notion of complete refinement. We then illustrated our technique with a simple case study based on holding food with chopsticks. The chopstick case study illustrated rather well the need for the domain theory discussed in general terms earlier.

We note that the conditions demanded of the controller and of the plant in our conditions for safe controller synthesis in (4), each relate the subsystem in question to the originating system (and only to the originating system). Thus they are completely symmetrical between the controller and plant and do not depend either on there being exactly two subsystems in play. Therefore, the result generalizes to a partition of the originating system into an arbitrary number of subsystems, each built in the same fashion, with some variables to which it has exclusive write access, and a larger set of variables to which it has read access.

The preceding remark is well illustrated by the chopstick case study, since after the initial decomposition into food (plant) and hand plus chopsticks (controller), we were able to repeat the decomposition of the hand plus chopsticks subsystem yielding a further separation into chopsticks (plant) and hand (controller), resulting in a three way partition of the original system.

We then gave a rather brief overview of continuous ASMs, arguing that for typical applications, such a brief description was usually sufficient, referring to a more through description elsewhere. In this context, we briefly discussed how the controller synthesis problem could be extended to the continuous formalism, illustrating it with a further elaboration of the chopsticks case study. We closed by discussing how mode rules that skip, generated by the decomposition process, could be discarded under certain circumstances, focusing on the more interesting continuous version of the theory for this.

Although we have targeted a very simple scenario, the ideas that we have explored have an applicability that is much wider than we have mentioned hitherto, especially in the context of today's hybrid and cyber-physical systems [7, 8, 9, 11]. In these, there is nowadays a strong tendency towards distributed solutions to problems describable in a global manner. So the initial global conception of the problem needs to be decomposed into a number of subsystems that co-operate to form the global solution. Not only are many of these problems intrinsically control problems anyway, making our approach directly applicable, but the abstract version of the decomposition technique that we have explored, tailored as it

is to the details of ASM rule scheduling, acts as a surrogate for a much wider gamut of problems and their solutions.

References

[1] Component-Based Software Engineering: International Symposia, Springer, LNCS, 1997 onwards.

[2] R. Banach, H. Zhu, W. Su, R. Huang, Continuous KAOS, ASM, and Formal Control System Design Across the Continuous/Discrete Modeling Interface: A Simple Train Stopping Application., Form. Asp. Comp. (2013). To appear.

[3] R. Banach, H. Zhu, W. Su, X. Wu, Continuous ASM, and a Pacemaker Sensing Fragment, in: Derrick, Fitzgerald, Gnesi, Khurshid, Leuschel, Reeves, Riccobene (Eds.), Proc. ABZ-12, volume 7316, Springer, LNCS, 2012, pp. 65–78.

[4] R. Banach, H. Zhu, W. Su, X. Wu, A Continuous ASM Modelling Approach to Pacemaker Sensing (2013). Submitted.

[5] R. Banach, H. Zhu, W. Su, X. Wu, Moded and Continuous ASM (2013). Submitted.

[6] E. Börger, R. Stärk, Abstract State Machines. A Method for High Level System Design and Analysis, Springer, 2003.

[7] A. Platzer, Logical Analysis of Hybrid Systems: Proving Theorems for Complex Dynamics, Springer, 2010.

[8] J. Sztipanovits, Model Integration and Cyber Physical Systems: A Semantics Perspective, in: Butler, Schulte (Eds.), Proc. FM-11, Springer, LNCS 6664, p.1, http://sites.lero.ie/download.aspx?f=Sztipanovits-Keynote.pdf, 2011. Invited talk, FM 2011, Limerick, Ireland.

[9] P. Tabuada, Verification and Control of Hybrid Systems: A Symbolic Approach, Springer, 2009.

[10] W. Walter, Ordinary Differential Equations, Springer, 1998.

[11] J. Willems, Open Dynamical Systems: Their Aims and their Origins. Ruberti Lecture, Rome, 2007. http://homes.esat.kuleuven.be/~jwillems/Lectures/2007/Rubertilecture.pdf.