A data-flow approach to test multi-agent ASMs

Alessandra Cavarra

Under consideration for publication in Formal Aspects of Computing

A data-flow approach to test multi-agent ASMs

Alessandra Cavarra

Oxford University Computing Laboratory
Wolfson Building, Parks Road, Oxford OX1 3QD UK

Abstract. This paper illustrates the theoretical basis of an approach to apply data flow testing techniques to Abstract State Machines. In particular, we focus on multi-agent ASMs extended with the **seq** construct for turbo ASMs. We explain why traditional data flow analysis can not simply be applied to ASMs: data flow coverage criteria are strictly based on the mapping between a program and its flow graph whereas in this context we are interested in tracing the flow of data between states in ASM runs as opposed to between nodes in a program's flow graph. We revise the classical concepts in data flow analysis taking into account the specific, parallel nature of ASMs, and define them on two levels: the syntactic (rule) level, and the computational (run) level. In particular, we analyze the role played by different types of terms in ASMs and deal with the problem of terms that are monitored by a given agent but controlled by another one, terms that are shared between several agents, and derived terms. We also discuss what consequences the use of the turbo ASM construct **seq** has on our analysis and revise the approach accordingly. Finally, we specify a family of ad hoc data flow coverage criteria for this class of ASMs and introduce a model checking-based approach to generate automatically test cases satisfying a given set of coverage criteria from ASM models.

Keywords: Model-based testing, Data-flow analysis, Abstract State Machines.

1. Introduction

The use of models for designing and testing software is currently one of the most noticeable industrial trends with significant impact on the development and testing processes.

Model-based testing (MBT) is a technique for generating a suite of test cases from a model encoding the intended behaviour of the system under test. This model can reside at various levels of abstraction. Model-based methods from object-oriented software engineering, formal methods, and other mathematical and engineering disciplines have now been successfully applied for automatic test case generation.

Modelling requires a substantial investment, and practical and scalable MBT solutions can help leverage this investment; the utility of models for generating test cases is a significant element in determining the cost effectiveness of producing formal or semi-formal specifications. In fact, the return of investment for

Correspondence and offprint requests to: Alessandra Cavarra, Oxford University Computing Laboratory, Wolfson Building, Parks Road, Oxford OX1 3QD UK e-mail: alessandra.cavarra@comlab.ox.ac.uk

model building should be evaluated not only in terms of enhanced understanding of the requirements of an application and how its architecture is designed, but also improved effectiveness and reduced effort and cost in test generation. Traditionally, the process of deriving tests tends to be unstructured, not reproducible, not documented, and bound to the ingenuity of single individuals. The existence of an artifact that explicitly encodes the intended behaviour of the system under test can help to reduce these problems in many ways—e.g. improved test cases and regression testing [UPL06]. Testers adopting this approach shift their attention from hand-crafting individual tests to the model of the system under test and a test generation infrastructure.

The model-based approach presented in this paper adopts specifications written using multi-agent Abstract State Machines [Gur95, BS03] as oracles for data flow analysis and testing. The choice of this formal language as a platform to define methods for generating test suites from high-level specifications is intentional and is due to the fact that, besides having evident theoretical foundations, and clear and precise semantics, ASMs have been successfully used in practice for design and analysis of complex hardware/software systems. We believe that our approach can help significantly in two ways: to validate large models and to test thoroughly their final implementation.

The original idea behind data flow testing is to track input variables through a program, following them as they are modified, until they are ultimately used to produce output values. In particular, the lifecycle of a piece of data is monitored to detect inappropriate definitions, use in predicates, computations and termination. Data flow coverage criteria are based on the intuition that one should not feel confident that a variable has been updated in a correct way at some stage in the program if no test exercises a computation path from the point where the variable is assigned a given value to the point where such value is subsequently used (in a predicate or computation).

In general, this idea fits well the ASM approach where in a given state a number of terms are updated and used to compute updates, provided that certain conditions are satisfied. Nevertheless, to our knowledge, data flow coverage criteria have never been defined for ASMs.

Methods using ASM models for automatic test generation exist in literature. In particular, in [GR01] and [GRR03] Gargantini et al. present a set of interesting coverage criteria together with two model checking-based tools (using, respectively, SMV [McM93] and SPIN [Hol97]) to generate test suites that accomplish the desired level of coverage. This approach focuses strictly on the structure of the ASM specification, and can be considered as the equivalent of control flow testing for ASMs. However, full coverage of all the rules and conditions in an ASM will not guarantee that all the possible patterns of usage of a term are exercised, therefore missing the opportunity to find potential errors in the way the term is processed.

Classical data flow adequacy criteria cannot be directly applied to ASMs since they are based on the one to one mapping between the code and its control flow graph. However, while control flow is usually explicitly defined for programs written in most programming languages, it is only implicit in ASMs and therefore the notion of flow graphs is not applicable to ASMs. We discuss here how to address this problem by modifying traditional data flow analysis concepts taking in consideration both the structure of the machine (i.e. its agents and rules) and its computations (i.e. the runs). Also, by varying the required combinations of definitions and uses of terms, we define a family of test data selection and adequacy criteria based on those illustrated in [RW85, FW88]. We also describe an approach based on model checking to generate automatically a collection of test cases satisfying a given set of coverage criteria.

The main purpose of this work is to set sound theoretical foundations of a data flow analysis method for ASMs. The definitions in [Cav08] and [Cav09] have been significantly revised, misconceptions in the treatment of controlled, monitored, shared, and derived terms both at the rule and the run level have been amended, and several imprecisions clarified. Moreover, we have extended the approach to include the **seq** construct for turbo ASMs [BS00].

The remainder of this paper is organized as follow. In Section 2 we give an introduction to multi-agent ASMs and term classification. In Section 3 we re-define the classical data-flow concepts in terms of ASMs and present a family of test data selection and adequacy criteria. In Section 4, we discuss the consequences of introducing the **seq** construct in a model, and give the definitions necessary for the approach to work also in a sequential environment. In section 5 we illustrate a model checking based approach to derive automatically a test suite satisfying the all-defs and all-uses coverage criteria. In section 5 we elucidate the approach by applying it to the Production Cell case study. Finally, in Section 6 we discuss our results, and related and future work.

2. Multi-Agent ASMs

In this section we give an introduction to the semantics of multi-agent Abstract State Machines.

A distributed ASM is given by a set of agents each of which is assigned a module (program) consisting of a finite number of so called transition rules of the following form:

if *Cond* **then** *Assignment*

where *Cond* is any expression (of first order logic) and *Assignment* is a finite set of function assignments[1] $f(t_1, \ldots, t_n) := t$. The states of ASMs are arbitrary structures, i.e. domains with predicates and functions defined on them. The collection of the types of the functions (and predicates) which can occur in a given ASM is called its signature. The computational meaning of an ASM $\mathcal{M}$ is that given any state S (of the signature of $\mathcal{M}$), for each transition rule such that *Cond* is true in S, all the assignments $f(t_1, \ldots, t_n) := t$ in the set *Assignments* of that rule are executed simultaneously, i.e. the value of function f at the given argument combination $t_1, \ldots, t_n$, computed in S, is changed to the value t which has been computed in S. The result of this computation step is a new state which differs from S only by some values for some of the functions where the 0-ary functions play the role of the usual programming variables. This definition covers the concept of "run" for basic ASMs. In the case of a distributed ASM, each agent fires its rules independently; the overall distributed run is a partially ordered set $(M, <)$ of rules applications–or moves–of its agents satisfying the following conditions: [*finite history*] each move has only finitely many predecessors; [*sequentiality of agents*] the set of moves of any single agent is linearly ordered by $<$; and [*coherence*] if m' is a maximal element in a finite initial segment X of moves $(M, <)$ and $Y = X \setminus \{m'\}$, then the state $S(X)$, obtained applying all the moves in X, is the result of applying move m' in state $S(Y)$.

An immediate corollary of the coherence condition is that all linearisations of an initial segment of a run result in the same final state. Observe that this definition does not describe how to construct partially ordered runs for a distributed ASM, therefore leaving one free to implement the described causal dependencies of certain local actions of otherwise independent agents [BS03].

ASMs usually come together with a set of integrity constraints (on the domains, functions, rules) and with initialization conditions representing assumptions on the intended computations.

We can view an abstract state as a memory that maps locations to values. Given a state S of a vocabulary $\mathcal{V}$, a *location* of S is a pair $l = (f, (t_1, \ldots, t_n))$, where f is an n-ary function name in $\mathcal{V}$, and $t_1, \ldots, t_n$ is an n-tuple of elements of S. The value $f^S(t_1, \ldots, t_n)$ is called the content of the location in S.

For the purpose of our analysis we will adopt the generic concept of *term* defined recursively as follows: (1) a variable is a term, (2) if f is a function name of arity n, and $t_1, \ldots, t_n$ are terms, then $f(t_1 \ldots, t_n)$ ($f(\bar{t})$ for short) is a term.

Let us now introduce an ASM classification of functions (or more generally terms) that has proved to be particularly convenient for applications specification. In a given ASM $\mathcal{M}$ of an agent $\mathcal{A}$, functions can be either static, i.e. never changing during any run of $\mathcal{M}$, or dynamic. Dynamic functions may change during a run of $\mathcal{M}$ as a consequence of assignments by $\mathcal{A}$ or assignments by the environment (i.e. by some other agent than $\mathcal{A}$). This results in the distinction of the following four subclasses of dynamic functions. *Controlled* functions (for $\mathcal{M}$) are dynamic functions which are directly updatable by and only by the rules of $\mathcal{M}$, i.e. functions which appear in the left hand side in assignments of rules of $\mathcal{M}$ and are not updatable by the environment. *Monitored* functions are dynamic functions that are directly updatable by and only by the environment. *Shared* functions are dynamic functions which are directly updatable by rules of $\mathcal{M}$ and by the environment. *Derived* functions are functions defined in terms of static and dynamic functions. Updatable functions are controlled or shared functions, non updatable functions are static, monitored or derived.

The following example illustrates these concepts. Please, notice that despite the fact that this is a simple, abstract model, it is appropriate for the purpose of this paper as it encloses most the features we are discussing here.

Example 2.1. Consider the following ASM $\mathcal{M}$ consisting of two agents, $\mathcal{A} = \{\mathcal{A}_1, \mathcal{A}_2\}$, each of which executes its own set of rules, where x, y, z, v are integer variables, and st is a function defined as $st : \mathcal{A} \rightarrow \{\text{N,C,S}\}$[2].

[1] We distinguish here between assignment (or definition) and update, where an update consists in the execution and effect of an assignment.

[2] Every agent can refer to its own copy of *state* as *state*(*Self*); however, when it is clear from the context, we omit *Self*.

$\mathcal{A}_1$

R_1:
1. **if** $st = N$ **then**
2. **if** $z \geq x$ **then**
3. $y := y\text{-}1$
4. $x := x\text{ -}1$
5. **else**
6 $x := x + 1$
7. $st := C$

R_2:
1. **if** $state = C$ **and** $x = 0$ **then**
2. $y := 0$
3. $st = S$

R_3:
1. **if** $st = C$ **and** $y > 0$ **then**
2. $x := x\text{ -}1$

$\mathcal{A}_2$

R_1:
1. **if** $st = N$ **then**
2. **if** $w > z$ **and** $z < x$ **then**
3. $y := y\text{ -}2$
4. $z := x + y$
5. **else**
6. $z := z - x$
7. **if** $st(A_1) = C$ **then**
8. $y := y\text{ -}1$
9. $st := C$
10. $v := w - v$
where $w \equiv x \times y$

R_2:
1. **if** $st = C$ **and** $z > y$ **then**
2. $z := z\text{ -}1$
3. **if** $z \leq 0$ **then**
4. $st := S$

The variables and functions in $\mathcal{A}_1$ are classified as follows:
– controlled functions: $st(\mathcal{A}_1)$, x
– shared functions: y
– monitored functions: z

The variables and functions in $\mathcal{A}_2$ are classified as follows:
– controlled functions: $st(\mathcal{A}_2)$, v, z
– shared functions: y
– monitored functions: $st(\mathcal{A}_1)$, x
– derived functions: w

□

3. Data flow analysis

In this section the main concepts of data flow analysis for ASMs, as defined in [Cav08] and later expanded in [Cav09], have been significantly revised to amend some imprecisions and errors, especially in the treatment of derived terms. We discuss the obstacles to adapting them to the ASM paradigm, and define ad-hoc coverage criteria for ASMs.

The goal of traditional data flow analysis is to detect data flow anomalies and errors (in the way data are processed and used). Data flow anomalies do not show that there is necessarily an error but indicate the possibility of program faults. For instance, very common anomalies found in a program are:

- *d-u anomalies*: they occur when a defined variable has not been referenced before it becomes undefined (e.g. out of scope or the program terminates). This anomaly usually indicates that the wrong variable has been defined or undefined.
- *d-d anomalies*: they indicate that the same variable is re-defined without being used, causing a hole in the scope of the first definition of the variable (this anomaly usually occurs because of misspelling).
- *u-r anomalies*: they occurs when an undefined variable is referenced. Most commonly u-r anomalies occur when a variable is referenced without having been initialised first.

Detecting the above anomalies is particularly difficult in the case of ASMs, where the code is distributed across several rules. Data flow analysis is very useful to uncover common types of programming errors such as typing errors, misspelling of names, misplacing of statements, or incorrect parameters.

However, it is not possible to apply directly data flow analysis to ASMs: the concept at the core of classical

P:
1. **if** $st = N$ **then**
2. **if** $z \geq x$ **then**
3. y := *y-1*
4. x := x *-1*
5. **else**
6 x := x *+1*
7. st := C

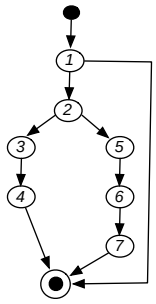

Fig. 1. A control flow graph

data flow analysis is the one-to-one mapping between a program and its flow graph. Given a program P, the flow graph associated to it is given by $G = (n_s, n_f, N, E)$, where N is the set of nodes labeled by the statements in P, n_s and n_f are respectively the start and finish nodes, and E is the set of edges representing possible flow of control between statements. Control flow graphs are built using the concept of programming primes, i.e. sequential, conditional, and loop statements.

While, in general, for any given program it is straightforward to derive its corresponding flow graph (see Figure 1), this is clearly not the case for ASMs, where the guards of all the rules in the model are evaluated simultaneously and, if true, all the corresponding assignments are executed simultaneously. Therefore there is no sequential flow between rules statements. (Although rule R_1 of agent $\mathcal{A}_1$ in Example 6 is syntactically equivalent to program **P** in Figure 1, semantically they are very different since all the statements in **P** will be executed sequentially, whereas the statements in R_1 will be executed simultaneously and any assignment will take effect at the next state.) We will see that the only exception to this is when the **seq** construct is used (see Section 4).

In the following, we provide our solution to this problem. We revise data flow concepts and provide ad-hoc definitions at two different levels: at the syntactic (rule) level and at the computational (run) level. We also provide a mapping between the concepts at different levels.

At a purely syntactic level, we introduce a sequential numbering for each rule in the ASM. To this purpose, we assume that every line of code will contain exactly one assignment or one the following constructs: **if** *cond*[3], **else**, **seq**. Moreover, without loss of generality we assume here that the defining equation of derived terms contains only controlled, shared, monitored, and static terms (this can simply be achieved by normalising the machine and substituting any further derived term it may contain with its defining equation).

3.1. Data flow concepts at the rule level

Terms can appear in different contexts in ASM rules: they can be updated, used in a guard, or used to compute a given value. However, as discussed above, terms can be used to define derived terms or, in the case of distributed ASMs an agent can use a term that is modified outside its scope, i.e. by another agent. In the following, we provide a number of definitions formalising the role ASM terms can play within rules.

Let $\mathcal{M}$ be a distributed ASM, and $\mathcal{A}$ the set of agents $\mathcal{A}_j$ each executing its own program $\mathcal{M}_j$.

Definition 3.1. Given a term $f(\bar{t})$, we say that it is *defined*—indicated as "def"—in a line k of a rule R_i of $\mathcal{M}_j$ if

- it is a controlled or shared term and it appears on the LHS of an assignment in R_i (i.e. the value of $f(\bar{t})$ may be modified as a result of firing R_i)
- it is a derived term with defining equation s, i.e. $f(\bar{t}) \equiv s$, and at least one of the sub-terms of s is defined in line k of R_i

We define the following sets at the agent level:

[3] Observe that **then** can be placed either in the same line as its corresponding **if**, or in the following one.

- $\text{def}^{R_i}_{\mathcal{A}_j}(f(\bar{t}))$ is the set of quadruples $(f(\bar{t}), k, R_i, \mathcal{A}_j)$ such that $f(\bar{t})$ is *defined* in line k of rule R_i in $\mathcal{M}_j$
- $\text{def}_{\mathcal{A}_j}(f(\bar{t}))$ contains all these quadruples across all the rules in $\mathcal{M}_j$, i.e.
  $\text{def}_{\mathcal{A}_j}(f(\bar{t})) = \bigcup_{R_i \in \mathcal{M}_j} \text{def}^{R_i}_{\mathcal{A}_j}(f(\bar{t}))$

At the global $\mathcal{M}$ level, we define the following set:
$$\text{def}_{\mathcal{M}}(f(\bar{t})) = \bigcup_{\mathcal{A}_j \in \mathcal{A}} \text{def}_{\mathcal{A}_j}(f(\bar{t}))$$
If $f(\bar{t})$ is a term controlled by an agent $\mathcal{A}_i$ its def sets at the agent and at the global level will coincide.

□

To perform our analysis, we need to be able to refer to a specific position, i.e. line, where a term is defined in a given rule. Notice that we cannot simply refer to an assignment as it is possible to have two identical assignments at different lines in the same rule (typically under different conditions). In order to retrieve a specific assignment, we introduce the function d which given a quadruple $(f(\bar{t}), k, R_i, \mathcal{A}_j)$ returns the assignment in line k of R_i. For instance, given the ASM in example 2.1, for $(z, 4, R_1, \mathcal{A}_2) \in \text{def}_{\mathcal{M}}(x)$, we obtain $d(z, 4, R_1, \mathcal{A}_2) = z := x + y$. Observe that, according to the above definition, in case $f(\bar{t})$ is a derived (and therefore not directly updatable) term the assignment returned by $d(f(\bar{t}), k, R_i, \mathcal{A}_j)$, for any $f(\bar{t}), k, R_i, \mathcal{A}_j \in \text{def}_{\mathcal{M}}(f(\bar{t}))$, will never refer to $f(\bar{t})$ itself, but only to a sub-term of the defining equation of $f(\bar{t})$. See for example, $\text{def}_{\mathcal{M}}(w)$ in example 3.1.

Work involving data flow analysis generally classifies each variable occurrence as being either a definition or a use. However, we distinguish between two substantially different types of uses. The first type directly affects the computation being performed or allows one to see the result of some earlier assignment. This is called "*c-use*". Of course, a c-use may indirectly affect the flow of control through the program. In contrast, the second type of use directly affects the flow of control through the program (by dictating which rules will be allowed to fire), and thereby may indirectly affect the computations performed. This is called "*p-use*" [RW85].

Definition 3.2. We say that a term $f(\bar{t})$ is in *predicate use*—indicated as "p-use"—in line k of rule R_i of $\mathcal{M}_j$ if

- it is a controlled, shared, or derived term and is used in a predicate shown in line k of R_i (i.e. $f(\bar{t})$ appears in a boolean condition in line k of R_i)
- it is a controlled, shared, or derived term and line k contains the **else** part of an **if-then-else** statement such that $f(\bar{t})$ is in p-use in the corresponding **if** part
- it appears in the defining equation of a derived term that is in p-use in line k of R_i

At the agent level, we define the following sets:

- $\text{p-use}^{R_i}_{\mathcal{A}_j}(f(\bar{t}))$ is the set of quadruples $(f(\bar{t}), k, R_i, \mathcal{A}_j)$ such that $f(\bar{t})$ is in *predicate use* in line k of rule R_i in $\mathcal{M}_j$
- $\text{p-use}_{\mathcal{A}_j}(f(\bar{t})) = \bigcup_{R_i \in \mathcal{M}_j} \text{p-use}^{R_i}_{\mathcal{A}_j}(f(\bar{t}))$

At the global $\mathcal{M}$ level, we define the following set:
$$\text{p-use}_{\mathcal{M}}(f(\bar{t})) = \bigcup_{\mathcal{A}_j \in \mathcal{A}} \text{p-use}_{\mathcal{A}_j}(f(\bar{t}))$$

□

In order to retrieve a specific predicate, we introduce the function p which given a quadruple $(f(\bar{t}), k, R_i, \mathcal{A}_j)$ returns the predicate in line k of R_i, or in the case of an **else** will produce the negation of the predicate shown in the line of the corresponding **if**. Observe that, according to the above definition, if $f(\bar{t})$ appears in the defining equation of a derived term g, the predicate $p(f(\bar{t}), k, R_i, \mathcal{A}_j)$ can contain either $f(\bar{t})$ or g. For instance, given $(y, 2, R_1, \mathcal{A}_2), (y, 5, R_1, \mathcal{A}_2) \in \text{p-use}_{\mathcal{M}}(y)$ in Example 3.1, $p(y, 2, R_1, \mathcal{A}_2) = w > z$ **and** $z < x$, and $p(y, 5, R_1, \mathcal{A}_2) =$**not** $(w > z$ **and** $z < x)$.

Definition 3.3. We say that a term $f(\bar{t})$ is in *computation use*—indicated as "c-use"—in line k of rule R_i of $\mathcal{M})_j$ if

- it is a controlled, shared, or derived term and is used in a computation in line k of R_i (i.e. it appears on the RHS of an assignment in line k of R_i)
- it appears in the defining equation of a derived term that is in c-use in line k of R_i
- it appears as an argument of a function displayed in any role in line k of rule R_i[4]

At the agent level, we define the following sets:

- $\text{c-use}^{R_i}_{\mathcal{A}_j}(f(\bar{t}))$ is the set of quadruples $(f(\bar{t}), k, R_i, \mathcal{A}_j)$ such that $f(\bar{t})$ is in *computation use* in line k of rule R_i in $\mathcal{M}_j$
- $\text{c-use}_{\mathcal{A}_j}(f(\bar{t})) = \bigcup_{R_i \in \mathcal{M}_|} \text{c-use}^{R_i}_{\mathcal{A}_j}(f(\bar{t}))$

At the global $\mathcal{M}$ level, we define the following set:

$$\text{c-use}_{\mathcal{M}}(f(\bar{t})) = \bigcup_{\mathcal{A}_j \in \mathcal{A}} \text{c-use}_{\mathcal{A}_j}(f(\bar{t}))$$

□

In order to refer to a specific computation, we introduce the function c which given a quadruple $(f(\bar{t}), k, R_i, \mathcal{A}_j)$ returns the assignment in line k of R_i where $f(\bar{t})$ is in c-use. Observe that, according to the above definition, if $f(\bar{t})$ appears in the defining equation of a derived term g, the RHS of the assignment $c(f(\bar{t}), k, R_i, \mathcal{A}_j)$ will not necessarily contain $f(\bar{t})$, but also g. For instance, given $(x, 10, R_1, \mathcal{A}_2) \in \text{c-use}_{\mathcal{M}}(x)$, $c(x, 10, R_1, \mathcal{A}_2) = v := w - v$.

Moreover, in general, we do not regard the terms used to define a derived function as in c-use (unless the derived term itself is used in a computation). This is because we only consider to be in c-use those terms that change the state of a the machine, whereas derived terms are not effectively part of the state of the ASM (see x and y in Example 3.1).

Example 3.1. Consider the ASM introduced in Example 2.1. Let us calculate the definition and use sets for the variables x, y, z, v, and w in the program of agent $\mathcal{A}_2$:

$def^{R_1}_{\mathcal{A}_2}(x) = \{\}$
$p\text{-}use^{R_1}_{\mathcal{A}_2}(x) = \{(x, 2, \mathcal{A}_2, R_1), (x, 5, \mathcal{A}_2, R_1)\}$
$c\text{-}use^{R_1}_{\mathcal{A}_2}(x) = \{(x, 4, \mathcal{A}_2, R_1), (x, 6, \mathcal{A}_2, R_1), (x, 10, \mathcal{A}_2, R_1)\}$

$def^{R_2}_{\mathcal{A}_2}(x) = \{\}$
$p\text{-}use^{R_2}_{\mathcal{A}_2}(x) = \{\}$
$c\text{-}use^{R_2}_{\mathcal{A}_2}(x) = \{\}$

$def^{R_1}_{\mathcal{A}_2}(y) = \{(y, 3, \mathcal{A}_2, R_1), (y, 8, \mathcal{A}_2, R_1)\}$
$p\text{-}use^{R_1}_{\mathcal{A}_2}(y) = \{(y, 2, \mathcal{A}_2, R_1), (y, 5, \mathcal{A}_2, R_1)\}$
$c\text{-}use^{R_1}_{\mathcal{A}_2}(y) = \{(y,3,\mathcal{A}_2,R_1),(y,4,\mathcal{A}_2,R_1),(y,8,\mathcal{A}_2,R_1),(y,10,\mathcal{A}_2,R_1)\}$

$def^{R_2}_{\mathcal{A}_2}(y) = \{\}$
$p\text{-}use^{R_2}_{\mathcal{A}_2}(y) = \{(y, 1, \mathcal{A}_2, R_2)\}$
$c\text{-}use^{R_2}_{\mathcal{A}_2}(y) = \{\}$

$def^{R_1}_{\mathcal{A}_2}(z) = \{(z, 4, \mathcal{A}_2, R_1), (z, 6, \mathcal{A}_2, R_1)\}$
$p\text{-}use^{R_1}_{\mathcal{A}_2}(z) = \{(z, 2, \mathcal{A}_2, R_1), (z, 5, \mathcal{A}_2, R_1)\}$
$c\text{-}use^{R_1}_{\mathcal{A}_2}(z) = \{(z, 4, \mathcal{A}_2, R_1), (z, 6, \mathcal{A}_2, R_1)\}$

$def^{R_2}_{\mathcal{A}_2}(z) = \{(z, 2, \mathcal{A}_2, R_2)\}$
$p\text{-}use^{R_2}_{\mathcal{A}_2}(z) = \{(z, 1, \mathcal{A}_2, R_2), (z, 3, \mathcal{A}_2, R_2)\}$
$c\text{-}use^{R_2}_{\mathcal{A}_2}(z) = \{(z, 2, \mathcal{A}_2, R_2)\}$

$def^{R_1}_{\mathcal{A}_2}(v) = \{(v, 10, \mathcal{A}_2, R_1)\}$
$p\text{-}use^{R_1}_{\mathcal{A}_2}(v) = \{\}$
$c\text{-}use^{R_1}_{\mathcal{A}_2}(v) = \{(v, 10, \mathcal{A}_2, R_1)\}$

$def^{R_2}_{\mathcal{A}_2}(v) = \{\}$
$p\text{-}use^{R_2}_{\mathcal{A}_2}(v) = \{\}$
$c\text{-}use^{R_2}_{\mathcal{A}_2}(v) = \{\}$

$def^{R_1}_{\mathcal{A}_2}(w) = \{\}$
$p\text{-}use^{R_1}_{\mathcal{A}_2}(w) = \{(w, 2, \mathcal{A}_2, R_1), (w, 5, \mathcal{A}_2, R_1)\}$
$c\text{-}use^{R_1}_{\mathcal{A}_2}(w) = \{(w, 10, \mathcal{A}_2, R_1)\}$

$def^{R_2}_{\mathcal{A}_2}(w) = \{\}$
$p\text{-}use^{R_2}_{\mathcal{A}_2}(w) = \{\}$
$c\text{-}use^{R_2}_{\mathcal{A}_2}(w) = \{\}$

[4] The only exception to this is for agents used as function parameters, since we do not perform data analysis on agents as such (see *st* in example 3.1).

In this example we could simply report line numbers as members of each set since the other three elements of the quadruples (i.e. term, agent, rule) are easily derived from the context. However, to avoid confusion, we show here complete quadruples.

Observe that variable y is in p-use in line 2 and 5 of R_1 because the derived variable w is in p-use in those lines. However, even though x and y are used "to compute" w they are not considered to be in c-use since, strictly speaking, this computation does not change the state of $\mathcal{A}_2$ (w is not actually part of the state of $\mathcal{A}_2$). On the other hand, since w is used in a computation in line 10 of R_1, both x and y are in computation use in it.

At the module (agent) level, we obtain:

$def_{\mathcal{A}_2}(st(\mathcal{A}_2)) = \{(st(\mathcal{A}_2),9,\mathcal{A}_2,R_1),(st(\mathcal{A}_1),3,\mathcal{A}_1,R_2)\}$
$p\text{-}use_{\mathcal{A}_2}(st(\mathcal{A}_2)) = \{(st(\mathcal{A}_2),1,\mathcal{A}_2,R_1),(st(\mathcal{A}_2),1,\mathcal{A}_2,R_2)\}$
$c\text{-}use_{\mathcal{A}_2}(st(\mathcal{A}_2)) = \{\}$

$def_{\mathcal{A}_2}(x) = \{\}$
$p\text{-}use_{\mathcal{A}_2}(x) = \{(x,2,\mathcal{A}_2,R_1),(x,5,\mathcal{A}_2,R_1)\}$
$c\text{-}use_{\mathcal{A}_2}(x) = \{(x,4,\mathcal{A}_2,R_1),(x,6,\mathcal{A}_2,R_1),(x,10,\mathcal{A}_2,R_1)\}$

$def_{\mathcal{A}_2}(y) = \{(y,3,\mathcal{A}_2,R_1),(y,8,\mathcal{A}_2,R_1)\}$
$p\text{-}use_{\mathcal{A}_2}(y) = \{(y,2,\mathcal{A}_2,R_1),(y,5,\mathcal{A}_2,R_1),(y,1,\mathcal{A}_2,R_2)\}$
$c\text{-}use_{\mathcal{A}_2}(y) = \{(y,3,\mathcal{A}_2,R_1),(y,4,\mathcal{A}_2,R_1),(y,8,\mathcal{A}_2,R_1),(y,10,\mathcal{A}_2,R_1)\}$

$def_{\mathcal{A}_2}(v) = \{(v, 10, \mathcal{A}_2, R_1)\}$
$p\text{-}use_{\mathcal{A}_2}(v) = \{\}$
$c\text{-}use_{\mathcal{A}_2}(v) = \{(v, 10, \mathcal{A}_2, R_1)\}$

$def_{\mathcal{A}_2}(z) = \{(z,4,\mathcal{A}_2,R_1),(z,6,\mathcal{A}_2,R_1),(z,2,\mathcal{A}_2,R_2)\}$
$p\text{-}use_{\mathcal{A}_2}(z) = \{(z,2,\mathcal{A}_2,R_1),(z,5,\mathcal{A}_2,R_1),(z,1,\mathcal{A}_2,R_2),(z,3,\mathcal{A}_2,R_2)\}$
$c\text{-}use_{\mathcal{A}_2}(z) = \{(z, 6, \mathcal{A}_2, R_1), (z, 2, \mathcal{A}_2, R_2)\}$

$def_{\mathcal{A}_2}(w) = \{\{(y,3,\mathcal{A}_2,R_1),(y,8,\mathcal{A}_2,R_1)\}$
$p\text{-}use_{\mathcal{A}_2}(w) = \{(w, 2, \mathcal{A}_2, R_1), (w, 5, \mathcal{A}_2, R_1)\}$
$c\text{-}use_{\mathcal{A}_2}(w) = \{(w, 10, \mathcal{A}_2, R_1)\}$

Finally, let us calculate the sets at the global level (i.e. putting together all the definition and use sets across the distributed ASM) for all the terms in the vocabulary of $\mathcal{M}$:

$def_{\mathcal{M}}(st(\mathcal{A}_1)) = \{(st(\mathcal{A}_1), 7, \mathcal{A}_1, R_1), (st(\mathcal{A}_1), 3, \mathcal{A}_1, R_2)\}$
$p\text{-}use_{\mathcal{M}}(st(\mathcal{A}_1)) = \{(st(\mathcal{A}_1), 1, \mathcal{A}_1, R_1), (st(\mathcal{A}_1), 1, \mathcal{A}_1, R_2), (st(\mathcal{A}_1), 1, \mathcal{A}_1, R_3), (st(\mathcal{A}_1), 7, \mathcal{A}_2, R_1)\}$
$c\text{-}use_{\mathcal{M}}(st(\mathcal{A}_1)) = \{\}$

$def_{\mathcal{M}}(st(\mathcal{A}_2)) = \{(st(\mathcal{A}_2), 9, \mathcal{A}_2, R_1), (st(\mathcal{A}_2), 4, \mathcal{A}_2, R_2)\}$
$p\text{-}use_{\mathcal{M}}(st(\mathcal{A}_2)) = \{(st(\mathcal{A}_2), 1, \mathcal{A}_2, R_1), (st(\mathcal{A}_2), 1, \mathcal{A}_2, R_2)\}$
$c\text{-}use_{\mathcal{M}}(st(\mathcal{A}_2)) = \{\}$

$def_{\mathcal{M}}(x) = \{(x, 4, \mathcal{A}_1, R_1), (x, 6, \mathcal{A}_1, R_1), (x, 2, \mathcal{A}_1, R_3)\}$
$p\text{-}use_{\mathcal{M}}(x) = \{(x, 2, \mathcal{A}_1, R_1), (x, 5, \mathcal{A}_1, R_1), (x, 1, \mathcal{A}_1, R_2), (x, 2, \mathcal{A}_2, R_1), (x, 5, \mathcal{A}_2, R_1)\}$
$c\text{-}use_{\mathcal{M}}(x) = \{(x, 4, \mathcal{A}_1, R_1), (x, 6, \mathcal{A}_1, R_1), (x, 2, \mathcal{A}_1, R_3), (x, 4, \mathcal{A}_2, R_1), (x, 6, \mathcal{A}_2, R_1), (x, 10, \mathcal{A}_2, R_1)\}$

$def_{\mathcal{M}}(y) = \{(y, 3, \mathcal{A}_1, R_1), (y, 3, \mathcal{A}_2, R_1), (y, 3, \mathcal{A}_2, R_1), (y, 8, \mathcal{A}_2, R_1)\}$
$p\text{-}use_{\mathcal{M}}(y) = \{(y, 1, \mathcal{A}_1, R_3), (y, 2, \mathcal{A}_2, R_1), (y, 5, \mathcal{A}_2, R_1), (y, 1, \mathcal{A}_2, R_2)\}$
$c\text{-}use_{\mathcal{M}}(y) = \{(y, 3, \mathcal{A}_1, R_1), (y, 4, \mathcal{A}_2, R_1), (y, 8, \mathcal{A}_2, R_1), (y, 10, \mathcal{A}_2, R_1)\}$

$def_{\mathcal{M}}(z) = \{(z, 4, \mathcal{A}_2, R_1), (z, 6, \mathcal{A}_2, R_1), (z, 2, \mathcal{A}_2, R_2)\}$
$p\text{-}use_{\mathcal{M}}(z) = \{(z, 2, \mathcal{A}_1, R_1), (z, 2, \mathcal{A}_2, R_1), (z, 5, \mathcal{A}_2, R_1), (z, 1, \mathcal{A}_2, R_2), (z, 3, \mathcal{A}_2, R_2)\}$
$c\text{-}use_{\mathcal{M}}(z) = \{(z, 6, \mathcal{A}_2, R_1), (z, 2, \mathcal{A}_2, R_2)\}$

$def_{\mathcal{M}}(w) = \{(x, 4, \mathcal{A}_1, R_1), (x, 6, \mathcal{A}_1, R_1), (x, 2, \mathcal{A}_1, R_3), (y, 3, \mathcal{A}_1, R_1), (y, 3, \mathcal{A}_2, R_1), (y, 3, \mathcal{A}_2, R_1), (y, 8, \mathcal{A}_2, R_1)\}$
$p\text{-}use_{\mathcal{M}}(w) = \{(w, 2, \mathcal{A}_2, R_1), (w, 5, \mathcal{A}_2, R_1)\}$
$c\text{-}use_{\mathcal{M}}(w) = \{(w, 10, \mathcal{A}_2, R_1)\}$

Finally, the def, p-use, and c-use sets of v at the global level is the same as the agent level.

3.2. Data flow concepts at the run level

After defining the possible roles of terms in a program, the next step in traditional data flow analysis consists in tracing through the program's control flow graph to search for paths from nodes where a variable is assigned

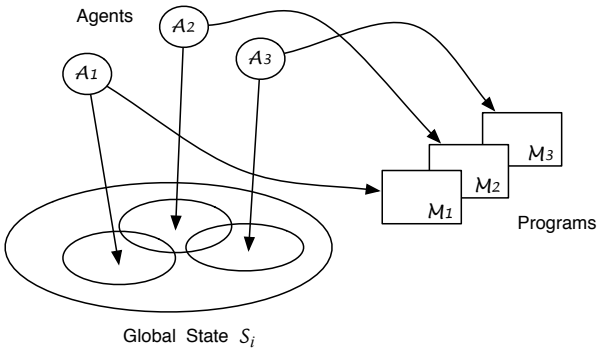

Fig. 2. Global state and partial views

a given value, to nodes where that value is used. Since, as explained above, in this context we cannot reason in terms of flow graphs, we need an alternative solution: we concentrate on ASM computations.

As discussed in Section 2, Abstract State Machines define a state-based computational model, where computations (runs) are finite or infinite sequences of states $\{s_i\}$, obtained from a given initial state $\{s_0\}$ by repeatedly executing transitions (rules) δ_i:

$$s_0 \xrightarrow{\delta_1} s_1 \xrightarrow{\delta_2} s_2 \ldots \xrightarrow{\delta_n} s_n$$

In the case of multi-agent ASMs, each agent is dynamically equipped with its own program operating on its own state, determining a partial view of the global system state (see Figure 2 adapted from [ITU00]).

In the following we describe how the concepts of definition and computation/predicate use at the rule level relate to ASM states. For this purpose, we need to revisit the definitions in the previous section in terms of ASM runs.

Definition 3.4. Let $f(\bar{t})$ be a term in the vocabulary of $\mathcal{M}$. We say that

- $f(\bar{t})$ is in def in a state s—indicated as "def$_s$"[5]—if the value of $f(\bar{t})$ was modified by the execution of the transition leading to s, i.e. $\exists\ (f(\bar{t}), k, R_i, \mathcal{A}_j) \in \text{def}_{\mathcal{M}}(f(\bar{t}))$ such that the value of $f(\bar{t})$ in s results from the execution of an assignment $d(f(\bar{t}), k, R_i, \mathcal{A}_j)$ in the transition leading to s. Terms are also considered to be in def$_s$ in the initial state of the machine (i.e. at initialisation time).
- $f(\bar{t})$ is in p-use in a state s—indicated as "p-use$_s$"—if $\exists\ (f(\bar{t}), k, R_i, \mathcal{A}_j) \in \text{p-use}_{\mathcal{M}}(f(\bar{t}))$ such that the predicate $p(f(\bar{t}), k, R_i, \mathcal{A}_j)$ evaluates to true in s.
- $f(\bar{t})$ is in c-use in a state s—indicated as "c-use$_s$"— if $\exists\ (f(\bar{t}), k, R_i, \mathcal{A}_j) \in \text{c-use}_{\mathcal{M}}(f(\bar{t}))$ such that the assignment $c(f(\bar{t}), k, R_i, \mathcal{A}_j)$ is executed when the transition leaving s is executed.

In particular, we say that term $f(\bar{t})$ is in def$_s$ (respectively c-use$_s$) in a state s w.r.t. $d(f(\bar{t}), k, R_i, \mathcal{A}_j)$ (resp. $c(f(\bar{t}), k, R_i, \mathcal{A}_j)$) if the assignment $d(f(\bar{t}), k, R_i, \mathcal{A}_j)$ (resp. $c(f(\bar{t}), k, R_i, \mathcal{A}_j)$) in rule R_i of the program of $\mathcal{A}_j$ is computed as a result of the execution of the transition leading to s. We say that term $f(\bar{t})$ is in p-use$_s$ in a state s w.r.t. $p(f(\bar{t}), k, R_i, \mathcal{A}_j)$ if the predicate $p(f(\bar{t}), k, R_i, \mathcal{A}_j)$ in rule R_i of the program of $\mathcal{A}_j$ is satisfied in s.

□

Example 3.2. To illustrate the above concepts let us consider an excerpt of a run of the ASM in Example 2.1 starting from the following state (see also Figure 3):

$$S_i = \begin{matrix}[\mathcal{A}_1\ ]\ st(\mathcal{A}_1) = N,\ x = 5,\ y = 7 \\ [\mathcal{A}_2\ ]\ st(\mathcal{A}_2) = N,\ z = 5,\ v = 2\end{matrix}$$

(Observe that, as a result, the value of the derived term $w = 35$.)
Since it exists a quadruple $(st(\mathcal{A}_1), 1, \mathcal{A}_1, R_1) \in \text{p-use}_{\mathcal{M}}(st(\mathcal{A}_1))$ such that the predicate $p(st(\mathcal{A}_1), 1, \mathcal{A}_1, R_1) =$

[5] Notice that the subscript here is not a parameter, but is used only to distinguish between a definition of a term at the rule level–labeled as def–and at the state level–def$_s$. Similarly, for p-use$_s$, and c-use$_s$.

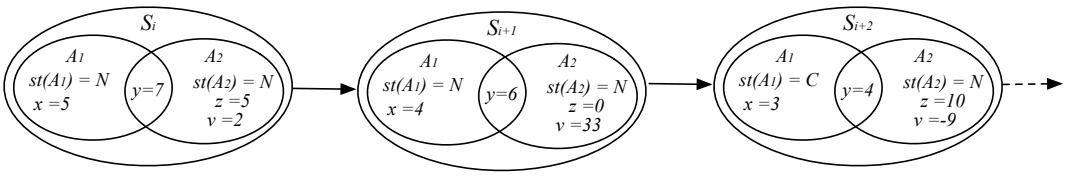

Fig. 3. A partial run of $\mathcal{M}$

$(st(\mathcal{A}_1) = N)$ evaluates to true in S_i, by definition $st(\mathcal{A}_1)$ is in *p-use*$_s$ in this state. Similarly, the predicates $p(st(\mathcal{A}_2), 1, \mathcal{A}_2, R_1)$, $p(x, 2, \mathcal{A}_1, R_1)$, $p(x/y/z/w, 5, \mathcal{A}_2, R_1)$ are satisfied in S_i and therefore $st(\mathcal{A}_2), x, y, z, w$ are in *p-use*$_s$ in it. This triggers rule R_1 in the module of $\mathcal{A}_1$ and R_1 in the module of $\mathcal{A}_2$; the assignments $c(x, 4, \mathcal{A}_1, R_1), c(y, 3, \mathcal{A}_1, R_1), c(z, 6, \mathcal{A}_2, R_1), c(x, 6, \mathcal{A}_2, R_1), c(v, 10, \mathcal{A}_2, R_1), c(w, 10, \mathcal{A}_2, R_1)$ are executed as part of the transition leaving S_i, thus by definition $x,\ y,\ z,\ v$ and w are in *c-use*$_s$ in S_i.

The state is modified as follows

$$S_{i+1} = \begin{matrix}[\mathcal{A}_1\ ]\ st(\mathcal{A}_1) = N,\ x = 4,\ y = 6 \\ [\mathcal{A}_2\ ]\ st(\mathcal{A}_2) = N,\ z = 0,\ v = 33\ (w = 24)\end{matrix}$$

Since the values of $x,\ y,\ z,\ v$ and w were modified by the execution of the transition incoming S_{i+1}, according to the definition they are in *def*$_s$ in S_{i+1}.

Finally, $p(st(\mathcal{A}_1), 1, \mathcal{A}_1, R_1)$, $p(x, 5, \mathcal{A}_1, R_1)$, $p(st(\mathcal{A}_2), 1, \mathcal{A}_2, R_1)$, and $p(x/y/z/w, 5, \mathcal{A}_2, R_1)$ are satisfied in S_{i+1}, the variables $st(\mathcal{A}_1), st(\mathcal{A}_2), x, y, z, w$ are in in *p-use*$_s$ in it.

□

Computations in which the value of a term is not modified play an important role in data-flow analysis. We say that a sub-run is *def-clear*$(f(\bar{t}))$ if it contains only states where $f(\bar{t})$ is not re-defined, i.e. the value of $f(\bar{t})$ is not updated in any of the states of the sub-run.

Definition 3.5. For each $(f(\bar{t}), k, R_i, \mathcal{A}_j) \in \text{def}_{\mathcal{M}}(f(\bar{t}))$, consider a state s such that $f(\bar{t})$ is in def_s in s w.r.t. $d(f(\bar{t}), k, R_i, \mathcal{A}_j)$. We define two sets of states:

- dpu$(s, f(\bar{t}))$ includes states s' such that there is a def-clear$(f(\bar{t}))$ sub-run from s to s' and $f(\bar{t})$ is in p-use$_s$ in s', i.e. there is a computation that starts with an assignment to $f(\bar{t})$, progresses while not reassigning to $f(\bar{t})$, and ends with a state where $f(\bar{t})$ is used within a predicate that holds true
- dcu$(s, f(\bar{t}))$ includes states s' such that there is a def-clear$(f(\bar{t}))$ sub-run from s to s' and $f(\bar{t})$ is in c-use$_s$ in s'.

3.3. Data flow coverage criteria

In this section we adapt the family of coverage criteria based on data flow information proposed by Rapps and Weyuker in [RW85] (and later extended in [FW88]). In general, such criteria require the definition of test data which cause the traversal of sub-paths from a variable definition to either some or all of the *p-uses*, *c-uses*, or their combination, or the traversal of at least one sub-path from each variable definition to every *p-use* and every *c-use* of that definition.

For each term $f(\bar{t})$ in the signature of $\mathcal{M}$ and for each state s such that $f(\bar{t})$ is in *def*$_s$ in s, we say that

- a test suite $\mathcal{T}$ satisfies the *all-defs* criterion if it includes one *def-clear*$(f(\bar{t}))$ run from s to some state in $dpu(s, f(\bar{t}))$ or in $dcu(s, f(\bar{t}))$
- a test suite $\mathcal{T}$ satisfies the *all-p-uses* (respectively, *all-c-uses*) criterion if it includes one *def-clear*$(f(\bar{t}))$ run from s to each state in $dpu(s, f(\bar{t}))$ (respectively, $dcu(s, f(\bar{t}))$)
- a test suite $\mathcal{T}$ satisfies the *all-c-uses/some-p-uses* if it includes one *def-clear*$(f(\bar{t}))$ run from s to each state in $dcu(s, f(\bar{t}))$, but if $dcu(s, f(\bar{t}))$ is empty, it includes at least one *def-clear*$(f(\bar{t}))$ run from s to some node in $dpu(s, f(\bar{t}))$
- a test suite $\mathcal{T}$ satisfies the *all-p-uses/some-c-uses* criterion if it includes one *def-clear*$(f(\bar{t}))$ run from s

to each state in $dpu(s, f(\bar{t}))$, but if $dpu(s, f(\bar{t}))$ is empty, it includes at least one *def-clear*$(f(\bar{t}))$ run from s to some node in $dcu(s, f(\bar{t}))$

- a test suite $\mathcal{T}$ satisfies the *all-uses* criterion if it includes one *def-clear*$(f(\bar{t}))$ run from s to each state in $dpu(s, f(\bar{t}))$ and to each state in $dcu(s, f(\bar{t}))$
- a test suite $\mathcal{T}$ satisfies the *all-du-paths* criterion if it includes all the cycle-free *def-clear*$(f(\bar{t}))$ runs from s to each state in $dpu(s, f(\bar{t}))$ and to each state in $dcu(s, f(\bar{t}))$

Empirical studies on traditional programming languages [Wey93, FW93] have shown that there is little difference in terms of the number of test cases sufficient to satisfy the least demanding criterion, *all-def*, and the most demanding criterion, *all-du-paths*. However, even if this should be the case also for ASM models, there is a hidden cost in satisfying the *all-du-paths* criterion, in that it is substantially more difficult to determine whether or not *all-du-paths* is actually satisfied due to the infeasibility problem: many definition-use (du-)paths can actually be non-executable, and it is frequently a difficult and time-consuming job to determine which du-paths are truly non-executable. For this reason, the most commonly adopted data flow criterion is the *all-uses*.

4. Turbo ASMs

In [BS00] basic ASMs have been extended to integrate the standard control constructs for sequentialization and iteration, and the notion of parameterized submachines into the classical ASM view of computations based on global state. Turbo ASMs are obtained from basic ASMs by applying infinitely often and in any order the operators of sequential composition, iteration, and submachine call.

We focus here on the **seq** operator which has been successfully applied to several problems [SSB01, BCR00]). It has the effect of combining simultaneous atomic updates of basic ASMs in a global state with sequential execution, i.e. all the statements in the scope of this construct will be executed sequentially but their effect will take place only in the following state. For example, consider the following rules:

R-par
1. **if** $x = y$
2. **then**
3. $x := x * 2$
4. $y := y + x$
5. **if** $x > 6 \wedge y > 10$
6. **then** $z := y$

R-seq
1. **if** $x = y$
2. **then**
3. **seq**
4. $x := x * 2$
5. $y := y + x$
6. **if** $x > 6 \wedge y > 10$
7. **then** $z := y$

If we initialise the variables as follows: $\{x = 4, y = 4, z = 1\}$, rule **R-par** will produce the state $\{x = 8, y = 8, z = 1\}$, whereas introducing the sequential construct **R-seq** will yield the state $\{x = 8, y = 12, z = 12\}$ since the value of y will be calculated according to the value of x assigned at line 4, and the predicate at line 6 will be evaluated according to the new values of x and y.

How does the use of the **seq** construct affect our data-flow analysis? We need to make some important observations here.

Firstly, as we have seen in the above example, in this context we actually have sequential flow within a rule, and therefore it is possible for a term to be defined and immediately used with the new value in the same rule.

Secondly, it is not possible to find a def-clear(x) path from any definition of x in other rules of the ASM to line 5 of **R-seq** (where x is in c-use), since the value of x will always be reassigned at line 4. Similarly, there is no def-clear(y) path to line 6 (where y is in p-use) from any other definition of y in the ASM besides the one at line 5. Therefore, in general we want to exclude these cases from our analysis. However, this is not necessarily the case when a term is defined and then used in a sequential environment: if the definition is in the scope of an **if** construct it may not actually be executed. For instance, in the example below if we initialise the variables as $\{x = 4, y = 4, z = 1\}$ the assignment of x at line 5 will not take place, and therefore it is possible to find a def-clear(x) from an assignment of x in another rule to its uses in line 6 and 7 in **R-seq1**.

R-seq1
1. **if** $x = y$
2. **then**
3. **seq**
4. **if** $x < 3$
5. **then** $x := x * 2$
6. $y := y + x$
7. **if** $x > 6 \wedge y > 10$
8. **then** $z := y$

We now need to formalise these concepts in our data flow analysis.

Definition 4.1. Given a definition $d(f(\bar{t}), k, R_i, \mathcal{A}_j)$, we say that a predicate $p(f(\bar{t}), k+l, R_i, \mathcal{A}_j)$ (respectively, a computation $c(f(\bar{t}), k+l, R_i, \mathcal{A}_j)$) with $l > 0$ *sequentially depends* on $d(f(\bar{t}), k, R_i, \mathcal{A}_j)$, if

(1) $d(f(\bar{t}), k, R_i, \mathcal{A}_j)$ and $p(f(\bar{t}), k+l, R_i, \mathcal{A}_j)$ (resp. $c(f(\bar{t}), k+l, R_i, \mathcal{A}_j)$) are in the context of the same **seq** construct, and

(2) $p(f(\bar{t}), k+l, R_i, \mathcal{A}_j)$ (resp. $c(f(\bar{t}), k+l, R_i, \mathcal{A}_j)$) is within the scope of at least as many **if** constructs as it is $d(f(\bar{t}), k, R_i, \mathcal{A}_j)$

□

Condition (2) guarantees that the term will always be redefined before reaching its use in the predicate/computation. Moreover, we request that $l > 0$ to exclude cases when a term is c-used and defined in the same line (e.g. the value of x in the computation in line 4 of **R-seq** is not yet affected by its definition even within the scope of **seq**).

While the definitions of terms in def, p-use and c-use at the rule and run level are valid also in case turbo ASMs with a **seq** operator, we need to modify the definition of dpu and dcu sets properly.

Definition 4.2. For each $(f(\bar{t}), k, R_i, \mathcal{A}_j) \in \text{def}_{\mathcal{M}}(f(\bar{t}))$, consider a state s such that $f(\bar{t})$ is in def_s in s w.r.t. $d(f(\bar{t}), k, R_i, \mathcal{A}_j)$. We define two sets of states:

- $\text{dpu}(s, f(\bar{t}))$ includes states s' such that there is a def-clear$(f(\bar{t}))$ sub-run from s to s', and $f(\bar{t})$ is in p-use_s in s' in a predicate that does not sequentially depend on any assignment of $f(\bar{t})$, i.e. there is a computation that starts with an assignment to $f(\bar{t})$, progresses while not reassigning to $f(\bar{t})$, and ends with a state where $f(\bar{t})$ is used within a predicate that does not sequentially depend on an update of $f(\bar{t})$.
- $\text{dcu}(s, f(\bar{t}))$ includes states s' such that there is a def-clear$(f(\bar{t}))$ sub-run from s to s', and $f(\bar{t})$ is in c-use_s in s' in a statement that does not sequentially depend on any assignment of $f(\bar{t})$.

□

With this new definition we exclude from the $\text{dpu}(s, f(\bar{t}))$ and $\text{dcu}(s, f(\bar{t}))$ sets those states where predicate and computation uses of $f(\bar{t})$ strictly depend on a value of $f(\bar{t})$ updated in a sequential context. The coverage criteria defined in section 3.3 now apply also to ASMs using the **seq** operator.

5. Generating test cases from ASMs

In the previous sections we have provided the theoretical basis for a data flow analysis of ASM specifications and defined a family of coverage criteria. We now address the problem of how to generate test suites satisfying a given set of such criteria for a multi-agent ASM model including the turbo ASM operator **seq**. Obviously, the hardest problem here is the need to reason in terms of all the possible computations of a given machine, i.e. to explore the state space of the machine. In the following, we elucidate an approach based on model checking.

Notice that given a quadruple $(f(\bar{t}), k, R_i, \mathcal{A}_j) \in \textit{def}_{\mathcal{M}}(f(\bar{t}))$ (respectively, $\textit{p-use}_{\mathcal{M}}$, $\textit{c-use}_{\mathcal{M}}(f(\bar{t}))$, for short we use here the notation $d^{f(\bar{t}),k}_{\mathcal{A}_j,R_i}$ (respect., $p^{f(\bar{t}),k}_{\mathcal{A}_j,R_i}$, $c^{f(\bar{t}),k}_{\mathcal{A}_j,R_i}$) in place of $d(f(\bar{t}), k, R_i, \mathcal{A}_j)$ (respect. $p(f(\bar{t}), k, R_i, \mathcal{A}_j)$, $c(f(\bar{t}), k, R_i, \mathcal{A}_j)$).

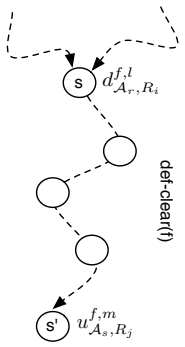

Fig. 4. A d-u pair

5.1. Model checking

In [Cav08] the method proposed by Hong et al. in [HCL$^+$03] was significantly modified in terms of Abstract State Machines. Lately, we have expanded it to multi-agent ASMs [Cav09]. In this paper we revise the approach, amend some errors, and extend it to allow for turbo ASMs with sequentiality. The underlying idea consists in representing data flow coverage criteria in temporal logic so that the problem of generating test suites satisfying a specific set of coverage criteria is reduced to the problem of finding witnesses for a set of temporal formulas. When the model checker determines that a formula with an existential path quantifier is true, it will find a computation path that demonstrates the success of the formula (witness). The capability of model checkers to construct witnesses [CGMZ95] allows for a fully automatic test generation process. In particular, in [HCL$^+$03] the model checker SMV [McM93] is used.

For this specific problem, Hong et al. introduce a subset of the existential fragment of CTL (ECTL) [CES86], called WCTL. An ECTL formula f is a WCTL formula if (i) f contains only **EX**, **EF**, and **EU**, where **E** ("for some path") is an existential path quantifier, **X** (next time), **F** (eventually), and **U** (until) are modal operators, and (ii) for every subformula of f of the form $f_1 \wedge \ldots \wedge f_n$, every conjunct f_i except at most one is an atomic proposition. For a full description refer to [HCL$^+$03].

Since, the original approach was designed for sequential programming languages, and therefore strongly based on control flow graphs, we had to modify it considerably in order to adapt it to multi-agent ASMs with sequentiality. Given any two agents $\mathcal{A}_r$ and $\mathcal{A}_s$ interacting in an ASM $\mathcal{M}$ and $(f(\bar{t}), l, R_i, \mathcal{A}_r) \in def_{\mathcal{M}}(f(\bar{t}))$ and $(f(\bar{t}), m, R_j, \mathcal{A}_s) \in p\text{-}use_{\mathcal{M}}(f(\bar{t})) \cup c\text{-}use_{\mathcal{M}}(f(\bar{t}))$, we say that an assignment $d^{f(\bar{t}),l}_{\mathcal{A}_r,R_i}$ (a definition of term $f(\bar{t})$ in line l of rule R_i of $\mathcal{A}_r$) and a use $u^{f(\bar{t}),m}_{\mathcal{A}_s,R_j}$ (a computation–$c^{f(\bar{t}),m}_{\mathcal{A}_s,R_j}$–or predicate–$p^{f(\bar{t}),m}_{\mathcal{A}_s,R_j}$–use of term $f(\bar{t})$ in line m of rule R_j of $\mathcal{A}_s$) constitute a *definition-use* pair (for short, "*du-pair*") $(d^{f(\bar{t}),l}_{\mathcal{A}_r,R_i}, u^{f(\bar{t}),m}_{\mathcal{A}_s,R_j})$ if there is a *def-clear*$(f(\bar{t}))$ path from state s to state s' such that $f(\bar{t})$ is in def_s in s with respect to $d^{f(\bar{t}),l}_{\mathcal{A}_r,R_i}$, and in *p/c-use*$_s$ in s' with respect to $u^{f(\bar{t}),m}_{\mathcal{A}_s,R_j}$ (see Figure 4).

Particular care must be taken in case the target computation/predicate use is within a sequential setting: a def-clear$(f(\bar{t}))$ path between a definition $d^{f(\bar{t}),l}_{\mathcal{A}_r,R_i}$ and a use $u^{f(\bar{t}),m}_{\mathcal{A}_s,R_j}$ can be found only if $u^{f(\bar{t}),m}_{\mathcal{A}_s,R_j}$ does not sequentially depend on an assignment of $f(\bar{t})$.

In general, the idea is to check that for some path eventually there is a state s where the value of $f(\bar{t})$ is modified, and from s there is a path in which $f(\bar{t})$ is not redefined until we reach a state where the value of $f(\bar{t})$ is used (in a predicate or computation). In ASM terms, this means that we are looking for a run such that at some point we reach a state where the predicate guarding the selected update $d^{f(\bar{t}),l}_{\mathcal{A}_r,R_i}$ is satisfied (therefore triggering a rule where the value of $f(\bar{t})$ is modified) and then all the predicates guarding updates of $f(\bar{t})$ hold false (so $f(\bar{t})$ is not redefined) until we reach a state where, in case of predicate use the predicate $p^{f(\bar{t}),m}_{\mathcal{A}_s,R_j}$ is satisfied, in case of computation use the condition guarding the update $c^{f(\bar{t}),m}_{\mathcal{A}_s,R_j}$ must be true.

This can be formalised as:

$$wctl(d^{f(\bar{t}),l}_{\mathcal{A}_r,R_i}, u^{f(\bar{t}),m}_{\mathcal{A}_s,R_j}) = \mathbf{EF}(sd^{f}_{\mathcal{A}_r,R_i} \wedge \mathbf{EXE}(\neg sd^{f}_{\mathcal{M}}\ \mathbf{U}\ su^{f}_{\mathcal{A}_s,R_j}))$$

where

$sd^f_{\mathcal{A}_r,R_i} \equiv guard(f(\bar{t}), l, R_i, \mathcal{A}_r)$

$sd^f_{\mathcal{M}} \equiv \bigvee_{g \in G(f(\bar{t}))} g$

$G(f(\bar{t})) \equiv \{guard(f(\bar{t}), k, R_i, \mathcal{A}_j) | (f(\bar{t}), k, R_i, \mathcal{A}_j) \in \mathrm{def}_{\mathcal{M}}(f(\bar{t}))\} \setminus \{su^f_{\mathcal{A}_s,R_j}\}$

$$su^f_{\mathcal{A}_s,R_j} = \begin{cases} p^{f(\bar{t}),m}_{\mathcal{A}_s,R_j} & \text{if } u^{f(\bar{t}),m}_{\mathcal{A}_s,R_j} \text{ is a predicate} \\ guard(f(\bar{t}), m, R_j, \mathcal{A}_s) & \text{if } u^{f(\bar{t}),m}_{\mathcal{A}_s,R_j} \text{ is a computation} \end{cases}$$

Finally, given a quadruple $(f(\bar{t}), k, R_i, \mathcal{A}_j) \in def_{\mathcal{M}} f(\bar{t})$ (respectively, $\in$ *c-use*$_{\mathcal{M}} f(\bar{t})$), the function *guard* $(f(\bar{t}), k, R_i, \mathcal{A}_j)$ returns the conjunction of all the predicates guarding the assignment $d^{f(\bar{t}),l}_{\mathcal{A}_r,R_i}$ (respect. $c^{f(\bar{t}),l}_{\mathcal{A}_r,R_i}$).

Observe that $sd^f_{\mathcal{A}_r,R_i}$ actually identifies the state before the term is updated. This is not a problem as we are guaranteed that, since the guard is true, in the next state the update will take place. A similar remark holds for $su^f_{\mathcal{A}_s,R_j}$ when $u^{f(\bar{t}),m}_{\mathcal{A}_s,R_j}$ is a computation. Moreover, we do not include the guard of $u^{f(\bar{t}),m}_{\mathcal{A}_s,R_j}$ in the disjunct $sd^f_{\mathcal{M}}$ even when there is an assignment of $f(\bar{t})$ within its scope because the value of $f(\bar{t})$ used in $u^{f(\bar{t}),m}_{\mathcal{A}_s,R_j}$ has not being redefined yet (see example below).

Finally, if $u^{f(\bar{t}),m}_{\mathcal{A}_s,R_j}$ sequentially depends on an assignment of $f(\bar{t})$, we do not calculate the formula $wctl(d^{f(\bar{t}),l}_{\mathcal{A}_r,R_i}, u^{f(\bar{t}),m}_{\mathcal{A}_s,R_j})$ as it is not possible to find a def-clear$(f(\bar{t}))$ computation between $d^{f(\bar{t}),l}_{\mathcal{A}_r,R_i}$ and $u^{f(\bar{t}),m}_{\mathcal{A}_s,R_j}$.

Example 5.1. Consider again the ASM in Example 2.1. Suppose we want to find a run covering the pair $(d^{y,3}_{\mathcal{A}_1,R_1}, p^{y,4}_{\mathcal{A}_2,R_1})$. This is equivalent to searching for witnesses for the following formula

$wctl(d^{y,4}_{\mathcal{A}_1,R_1}, c^{y,4}_{\mathcal{A}_2,R_1}) =$
EF $((st(\mathcal{A}_1) = N \wedge z \geq x) \wedge$
EXE $(\neg((st(\mathcal{A}_1) = N \wedge z \geq x) \vee (st(\mathcal{A}_1) = C \wedge x = 0) \vee (st(\mathcal{A}_2) = C))$
U $(st(\mathcal{A}_2) = N \wedge (w > z \wedge z < x))))$

We do not include $guard(y, 3, R_1, \mathcal{A}_1)$ in the disjunction because the definition in its scope will only take place at the next state and therefore the value used in $c^{y,4}_{\mathcal{A}_2,R_1}$ is that assigned at $d^{y,4}_{\mathcal{A}_1,R_1}$, as desired.

Notice that if $d^{y,3}_{\mathcal{A}_2,R_1}$ and $c^{y,4}_{\mathcal{A}_2,R_1}$ were in the scope of a **seq** construct, we would have not computed the wctl formula. On the other hand, it would still be necessary to take into consideration the formula $wctl(d^{y,4}_{\mathcal{A}_1,R_1}, c^{y,3}_{\mathcal{A}_2,R_1})$.

□

Let us now describe how to generate a set of test sequences satisfying the *all-defs* and *all-uses* criteria for a set of pairs $(d^{f(\bar{t}),l}_{\mathcal{A}_r,R_i}, u^{f(\bar{t}),m}_{\mathcal{A}_s,R_j})$. Basically, we associate a formula $wctl(d^{f(\bar{t}),l}_{\mathcal{A}_r,R_i}, u^{f(\bar{t}),m}_{\mathcal{A}_s,R_j})$ with a pair $(d^{f(\bar{t}),l}_{\mathcal{A}_r,R_i}, u^{f(\bar{t}),m}_{\mathcal{A}_s,R_j})$ for every $(f(\bar{t}), l, R_i, \mathcal{A}_r) \in def_{\mathcal{M}}(f(\bar{t}))$ and $(f(\bar{t}), m, R_j, \mathcal{A}_s) \in use_{\mathcal{M}}(f(\bar{t}))$, and characterise each coverage criterion in terms of *witness-sets* for the formulas $wctl(d^{f(\bar{t}),l}_{\mathcal{A}_r,R_i}, u^{f(\bar{t}),m}_{\mathcal{A}_s,R_j})$. This reduces the problem of generating a test suite to the problem of finding a witness-set for a set of WCTL formulas. We say that Π is a witness-set for a set of WCTL formulas F if it consists of a set of finite paths such that, for every formula f in F there is a finite path π in Π that is a witness for f.

All-defs A test suite $\mathcal{T}$ satisfies the *all-defs* coverage criterion if, given any two agents $\mathcal{A}_r$ and $\mathcal{A}_s$ in $\mathcal{M}$, for every $(f(\bar{t}), l, R_i, \mathcal{A}_r) \in def_{\mathcal{M}}(f(\bar{t}))$ and some $(f(\bar{t}), m, R_j, \mathcal{A}_s) \in use_{\mathcal{M}}(f(\bar{t}))$, there is a test sequence in $\mathcal{T}$ covering some *def-clear*(f) run from a state where $f(\bar{t})$ is in def_s w.r.t. $d^{f(\bar{t}),l}_{\mathcal{A}_r,R_i}$ to a state where $f(\bar{t})$ is in *p/c-use*$_s$ w.r.t. $u^{f(\bar{t}),m}_{\mathcal{A}_s,R_j}$.

A test suite $\mathcal{T}$ satisfies the *all-defs* coverage criterion if and only if it is a witness-set for

$$\{ \bigvee_{(f(\bar{t}),m,R_j,\mathcal{A}_s)\in use_{\mathcal{M}}(f(\bar{t}))} \mathbf{wctl}(d^{f(\bar{t}),l}_{\mathcal{A}_r,R_i}, u^{f(\bar{t}),m}_{\mathcal{A}_s,R_j}) \mid (f(\bar{t}),l,R_i,\mathcal{A}_r)\in def_{\mathcal{M}}(f(\bar{t})), \text{ for every } f(\bar{t}) \text{ in } \mathcal{V}(\mathcal{M}), \text{ and } \mathcal{A}_r, \mathcal{A}_s \text{ in } \mathcal{M}\}$$

where
$use_{\mathcal{M}}(f(\bar{t})) = \{(f(\bar{t}),l,R_j,\mathcal{A}_r) \mid (f(\bar{t}),l,R_j,\mathcal{A}_r) \in p\text{-}use_{\mathcal{M}}(f(\bar{t})) \bigcup c\text{-}use_{\mathcal{M}}(f(\bar{t}))$ and $(f(\bar{t}),l,R_j,\mathcal{A}_r)$ does not sequentially depend on any assignment of $f(\bar{t})\}$

All-uses A test suite $\mathcal{T}$ satisfies the *all-uses* coverage criterion if, given any two agents $\mathcal{A}_r$ and $\mathcal{A}_s$ in $\mathcal{M}$, for every $(f(\bar{t}),l,R_i,\mathcal{A}_r) \in def_{\mathcal{M}}(f(\bar{t}))$ and every $(f(\bar{t}),m,R_j,\mathcal{A}_s) \in use_{\mathcal{M}}(f(\bar{t}))$, there is a test sequence in $\mathcal{T}$ covering some *def-clear*(f) run from a state where $f(\bar{t})$ is in def_s w.r.t. $d^{f(\bar{t}),l}_{\mathcal{A}_r,R_i}$ to a state where $f(\bar{t})$ is in $p/c\text{-}use_s$ w.r.t. $u^{f(\bar{t}),m}_{\mathcal{A}_s,R_j}$.

A test suite $\mathcal{T}$ satisfies the *all-uses* coverage criterion if and only if it is a witness-set for

$$\{\mathbf{wctl}(d^{f(\bar{t}),l}_{\mathcal{A}_r,R_i}, u^{f(\bar{t}),m}_{\mathcal{A}_s,R_j}) \mid (f(\bar{t}),l,R_i,\mathcal{A}_r)\in def_{\mathcal{M}}(f(\bar{t})), (f(\bar{t}),m,R_j,\mathcal{A}_s)\in use_{\mathcal{M}}(f(\bar{t})) \text{ for every } f(\bar{t}) \text{ in } \mathcal{V}(\mathcal{M}), \text{ and } \mathcal{A}_r, \mathcal{A}_s \text{ in } \mathcal{M}\}$$

Observe that, in the worst case, the number of formulas can be quadratic in the size of statements in the ASM.

Remark. In order to reduce testing time and effort, both the *all-defs* and the *all-uses* criteria can be restricted to specific agents, i.e. instead of searching for runs from the updates of a term across all the agents to all its uses, we could search for runs from the updates of a term in a specific subset of agents to its use in all the agents, or viceversa, from all the updates of the term across all the agents to its uses in a subset of agents.

Example 5.2. Consider again the ASM $\mathcal{M}$ defined in Example 2.1. Suppose we want to satisfy the *all-defs* criterion for variable x. This will involve finding a witness for the following three disjunctions

$$\begin{aligned}
&\{(wctl(d^{x,4}_{\mathcal{A}_1,R_1}, p^{x,2}_{\mathcal{A}_1,R_1}) \vee wctl(d^{x,4}_{\mathcal{A}_1,R_1}, p^{x,5}_{\mathcal{A}_1,R_1}) \vee wctl(d^{x,4}_{\mathcal{A}_1,R_1}, p^{x,1}_{\mathcal{A}_1,R_2}) \vee wctl(d^{x,4}_{\mathcal{A}_1,R_1}, p^{x,2}_{\mathcal{A}_2,R_1}) \vee \\
&\quad wctl(d^{x,4}_{\mathcal{A}_1,R_1}, p^{x,5}_{\mathcal{A}_2,R_1}) \vee wctl(d^{x,4}_{\mathcal{A}_1,R_1}, c^{x,4}_{\mathcal{A}_1,R_1}) \vee wctl(d^{x,4}_{\mathcal{A}_1,R_1}, c^{x,6}_{\mathcal{A}_1,R_1}) \vee wctl(d^{x,4}_{\mathcal{A}_1,R_1}, c^{x,2}_{\mathcal{A}_1,R_3}) \vee \\
&\quad wctl(d^{x,4}_{\mathcal{A}_1,R_1}, c^{x,4}_{\mathcal{A}_2,R_1}) \vee wctl(d^{x,4}_{\mathcal{A}_1,R_1}, c^{x,6}_{\mathcal{A}_2,R_1}) \vee wctl(d^{x,4}_{\mathcal{A}_1,R_1}, c^{x,10}_{\mathcal{A}_2,R_1})), \\
&\quad (wctl(d^{x,6}_{\mathcal{A}_1,R_1}, p^{x,2}_{\mathcal{A}_1,R_1}) \vee wctl(d^{x,6}_{\mathcal{A}_1,R_1}, p^{x,5}_{\mathcal{A}_1,R_1}) \vee wctl(d^{x,6}_{\mathcal{A}_1,R_1}, p^{x,1}_{\mathcal{A}_1,R_2}) \vee wctl(d^{x,6}_{\mathcal{A}_1,R_1}, p^{x,2}_{\mathcal{A}_2,R_1}) \vee \\
&\quad wctl(d^{x,6}_{\mathcal{A}_1,R_1}, p^{x,5}_{\mathcal{A}_2,R_1}) \vee wctl(d^{x,6}_{\mathcal{A}_1,R_1}, c^{x,4}_{\mathcal{A}_1,R_1}) \vee wctl(d^{x,6}_{\mathcal{A}_1,R_1}, c^{x,6}_{\mathcal{A}_1,R_1}) \vee wctl(d^{x,6}_{\mathcal{A}_1,R_1}, c^{x,2}_{\mathcal{A}_1,R_3}) \vee \\
&\quad wctl(d^{x,6}_{\mathcal{A}_1,R_1}, c^{x,4}_{\mathcal{A}_2,R_1}) \vee wctl(d^{x,6}_{\mathcal{A}_1,R_1}, c^{x,6}_{\mathcal{A}_2,R_1}) \vee wctl(d^{x,6}_{\mathcal{A}_1,R_1}, c^{x,10}_{\mathcal{A}_2,R_1})), \\
&\quad (wctl(d^{x,2}_{\mathcal{A}_2,R_3}, p^{x,2}_{\mathcal{A}_1,R_1}) \vee wctl(d^{x,2}_{\mathcal{A}_2,R_3}, p^{x,5}_{\mathcal{A}_1,R_1}) \vee wctl(d^{x,2}_{\mathcal{A}_2,R_3}, p^{x,1}_{\mathcal{A}_1,R_2}) \vee wctl(d^{x,2}_{\mathcal{A}_2,R_3}, p^{x,2}_{\mathcal{A}_2,R_1}) \vee \\
&\quad wctl(d^{x,2}_{\mathcal{A}_2,R_3}, p^{x,5}_{\mathcal{A}_2,R_1}) \vee wctl(d^{x,2}_{\mathcal{A}_2,R_3}, c^{x,4}_{\mathcal{A}_1,R_1}) \vee wctl(d^{x,2}_{\mathcal{A}_2,R_3}, c^{x,6}_{\mathcal{A}_1,R_1}) \vee wctl(d^{x,2}_{\mathcal{A}_2,R_3}, c^{x,2}_{\mathcal{A}_1,R_3}) \vee \\
&\quad wctl(d^{x,2}_{\mathcal{A}_2,R_3}, c^{x,4}_{\mathcal{A}_2,R_1}) \vee wctl(d^{x,2}_{\mathcal{A}_2,R_3}, c^{x,6}_{\mathcal{A}_2,R_1}) \vee wctl(d^{x,2}_{\mathcal{A}_2,R_3}, c^{x,10}_{\mathcal{A}_2,R_1})) \\
&\}
\end{aligned}$$

If we want to satisfy the *all-uses* criterion for the term z we need to find a witness-set for the set of formulas

$$\begin{aligned}
&\{wctl(d^{z,4}_{\mathcal{A}_2,R_1}, p^{z,2}_{\mathcal{A}_1,R_1}), wctl(d^{z,4}_{\mathcal{A}_2,R_1}, p^{z,2}_{\mathcal{A}_2,R_1}), wctl(d^{z,4}_{\mathcal{A}_2,R_1}, p^{z,5}_{\mathcal{A}_2,R_1}), wctl(d^{z,4}_{\mathcal{A}_2,R_1}, p^{z,1}_{\mathcal{A}_2,R_2}), wctl(d^{z,4}_{\mathcal{A}_2,R_1}, p^{z,3}_{\mathcal{A}_2,R_2}), \\
&\quad wctl(d^{z,4}_{\mathcal{A}_2,R_1}, c^{z,6}_{\mathcal{A}_2,R_1}), wctl(d^{z,4}_{\mathcal{A}_2,R_1}, c^{z,2}_{\mathcal{A}_2,R_2}), wctl(d^{z,6}_{\mathcal{A}_2,R_1}, p^{z,2}_{\mathcal{A}_1,R_1}), wctl(d^{z,6}_{\mathcal{A}_2,R_1}, p^{z,2}_{\mathcal{A}_2,R_1}), wctl(d^{z,6}_{\mathcal{A}_2,R_1}, p^{z,5}_{\mathcal{A}_2,R_1}), \\
&\quad wctl(d^{z,6}_{\mathcal{A}_2,R_1}, p^{z,1}_{\mathcal{A}_2,R_2}), wctl(d^{z,6}_{\mathcal{A}_2,R_1}, p^{z,3}_{\mathcal{A}_2,R_2}), wctl(d^{z,6}_{\mathcal{A}_2,R_1}, c^{z,6}_{\mathcal{A}_2,R_1}), wctl(d^{z,6}_{\mathcal{A}_2,R_1}, c^{z,2}_{\mathcal{A}_2,R_2}), wctl(d^{z,2}_{\mathcal{A}_2,R_2}, p^{z,2}_{\mathcal{A}_1,R_1}), \\
&\quad wctl(d^{z,2}_{\mathcal{A}_2,R_2}, p^{z,2}_{\mathcal{A}_2,R_1}), wctl(d^{z,2}_{\mathcal{A}_2,R_2}, p^{z,5}_{\mathcal{A}_2,R_1}), wctl(d^{z,2}_{\mathcal{A}_2,R_2}, p^{z,1}_{\mathcal{A}_2,R_2}), wctl(d^{z,2}_{\mathcal{A}_2,R_2}, p^{z,3}_{\mathcal{A}_2,R_2}), wctl(d^{z,2}_{\mathcal{A}_2,R_2}, c^{z,6}_{\mathcal{A}_2,R_1}), \\
&\quad wctl(d^{z,2}_{\mathcal{A}_2,R_2}, c^{z,2}_{\mathcal{A}_2,R_2})\}
\end{aligned}$$

□

6. A running example

As mentioned before, the main objective of this paper is to set the theoretical basis for a data-flow approach for Abstract State Machines. Consequently, the implementation and evaluation of the approach are beyond the scope of this work (see section 7). In this section we briefly discuss the overall testing process we propose and apply our method to the Production Cell case study. The choice of this example is motivated by the fact that it is a real system, as opposed to a "purpose-made" one, and that the ASM specification for this system originally proposed by Börger and Mearelli in [BM97] has been translated in SMV in [Win97, Win].

A testing process. In the following we briefly outline the overall testing process we propose in the form of use cases, where every T# indicates an action taken by the system, U# indicates a user action, and UT# indicates a user-tool interaction.

U0. The user edits the model into the chosen ASM tool.

T1. The ASM model is automatically translated into an equivalent SMV model.

U2. The user selects the set of terms to test, and possibly the set of agents where to perform the analysis (with the option of selecting all variables/agents)

U3. The user selects the criterion to satisfy

T4. The tool traverses the model, collects all the definitions and uses of a term, and automatically generates the formulas to satisfy. It will finally feed the SMV model checker with the model and the formulas

T5. SMV will return the test cases (witnesses) satisfying the chosen criterion.

UT6. The user will then run the test cases using the given ASM tool and will check that it behaves according to its requirements.

The Production Cell case study. "*... the production cell is composed of two conveyor belts, a positioning table, a two-armed robot, a press, and a traveling crane. Metal plates inserted in the cell via the feed belt are moved to the press. There, they are forged and then brought out of the cell via the other belt and the crane.*".

The system is modeled as a distributed ASM $\mathcal{M}$ with six agents—the Feed Belt (FB), the Robot, the Press, the Deposit Belt (DB), the Traveling Crane (TC), the Elevating Rotary Table (ERT)—composing the production cell, and working together concurrently. Each of the agents represents a sequential process which can execute its rules as soon as they become enabled. The sequential control of each agent is formalized using a function currPhase: Agent → Phase which contains at each moment the current phase of the agent. When clear from the context, we write currPhase in place of currPhase(Self). We refer the reader to [BM97] for further details.

Feed Belt [FB]

(R_1) FB NORMAL

```
1. if currPhase = NormalRun and
     PieceInFeedBeltLightBarrier
2. then FeedBeltFree := True
3.        if TableReadyForLoading
4.        then currPhase := CriticalRun
5.        else
6.             currPhase := Stopped
```

(R_2) FB STOPPED

```
1. if currPhase = Stopped and
     TableReadyForLoading
2. then currPhase := CriticalRun
```

(R_3) FB CRITICAL

```
1. if currPhase = CriticalRun and
     not PieceInFeedBeltLightBarrier
2. then currPhase := NormalRun
3.        TableLoaded := True
```

where
TableReadyForLoading ≡ (currPhase(ERT) = StoppedInLoadPos) **or not** TableLoaded

Elevating Rotary Table [ERT]

(R_1) WAITING LOAD
1. **if** currPhase = StoppedInLoadPosition **and**
   TableLoaded
2. **then** currPhase := MovingToUnloadPosition

(R_2) MOVING UNLOAD
1. **if** currPhase = MovingToUnloadPosition **and**
   UnloadPositionReached
2. **then** currPhase := StoppedInUnloadPosition

Travelling Crane [TC]

(R_1) WAITING(DB)
1. **if** currPhase= WaitingToUnloadDepositBelt **and**
   PieceAtDepositBeltEnd
2. **then** currPhase := UnloadingDepositBelt

(R_2) UNLOADING(DB)
1. **if** currPhase= UnloadingDepositBelt **and**
   UnloadingDepositBeltCompleted
2. **then** currPhase := MovingToLoadFeedBeltPos
3. PieceAtDepositBeltEnd := false

(R_3) MOVING(FB)
1. **if** currPhase= MovingToLoadFeedBeltPosition **and**
   LoadFeedBeltPosReached
2. **then** currPhase:= WaitingToLoadFeedBelt

(R_3) WAITING UNLOAD
1. **if** currPhase = StoppedInUnloadPosition **and**
   not TableLoaded
2. **then** currPhase := MovingToLoadPosition

(R_4) MOVING LOAD
1. **if** currPhase = MovingToLoadPosition **and**
   LoadPositionReached
2. **then** currPhase := StoppedInLoadPosition

(R_4) WAITING(FB)
1. **if** currPhase= WaitingToLoadFeedBelt **and**
   FeedBeltFree
2. **then** currPhase := LoadingFeedBelt

(R_5) LOADING(FB)
1. **if** currPhase = LoadingFeedBelt **and**
   LoadingFeedBeltCompleted
2. **then** currPhase := MovingToUnloadDepositBeltPos
3. FeedBeltFree := false

(R_6) MOVING(DB)
1. **if** currPhase= MovingToUnloadDepositBeltPos **and**
   UnloadDepositBeltPosReached
2. **then** currPhase := WaitingToUnloadDepositBelt

ROBOT [R]

($R_1|R_4|R_7|R_{10}$) WAITING[TABLE-UL|PRESS-UL|DEPBELT-L|PRESS-L]
1. **if** currPhase = WaitingIn[UnloadTable|UnloadPress|LoadDepBelt|LoadPress]Pos **and**
   [Table|Press|DepositBelt|Press)ReadyFor(Unloading|Unloading|Loading|Loading]
2. **then** currPhase := UnloadingTable|UnloadingPress|LoadingDepBelt|LoadingPress

($R_2|R_5|R_8|R_{11}$) ACTION[TABLE-UL|PRESS-UL|DEPBELT-L|PRESS-L]
1. **if** currPhase = UnloadingTable|UnloadingPress|LoadingDepBelt|LoadingPress **and**
   [UnloadingTable|UnloadingPress|LoadingDepBelt|LoadingPress]Completed
2. **then** currPhase:=MovingTo[UnloadPress|LoadDepBelt|LoadPress|UnloadTable]Pos
3. TableLoaded|PressLoaded|DepositBeltReadyForLoading|PressLoaded := false|false|false|true

($R_3|R_6|R_9|R_{12}$) MOVING[TABLE-UL|PRESS-UL|DEPBELT-L|PRESS-L]
1. **if** currPhase = MovingTo[UnloadPress|LoadDepBelt|LoadPress|UnloadTable]Pos **and**
   [UnloadPress|LoadDepBelt|LoadPress|UnloadTable]PosReached
2. **then** currPhase:=WaitingIn[UnloadPress|LoadDepBelt|LoadPress|UnloadTable]Pos

where
TableReadyForUnloading ≡ ((currPhase(ERT) = StoppedInUnloadPosition) **and** TableLoaded)
PressReadyForUnloading ≡ ((currPhase(Press) = OpenForUnloading) **and** PressLoaded)
PressReadyForLoading ≡ ((currPhase(Press) = OpenForLoading) **and not** PressLoaded)

Deposit Belt [DB]

(R_1) DB NORMAL
1. **if** currPhase = NormalRun **and**
   PieceInDepositBeltLightBarrier
2. **then** currPhase := CriticalRun

(R_2) DB CRITICAL
1. **if** currPhase = CriticalRun **and**
   not PieceInDepositBeltLightBarrier
2. **then** currPhase := Stopped
3. DepositBeltReadyForLoading := true
4. PieceAtDepositBeltEnd := true

(R_3) DB STOPPED
1. **if** currPhase = Stopped **and**
   not PieceAtDepositBeltEnd
2. **then** currPhase := NormalRun

Press [P]

(R_1) WAITING UNLOAD
1. **if** currPhase = OpenForUnloading **and** **not** PressLoaded
2. **then** currPhase := MovingToMiddlePosition

(R_2) MOVING TO MIDDLE
1. **if** currPhase = MovingToMiddlePosition **and** MiddlePosition
2. **then** currPhase := OpenForLoading

(R_3) WAITING LOAD
1. **if** currPhase = OpenForLoading **and** PressLoaded
2. **then** currPhase := MovingToTopPosition

(R_4) MOVING TO UPPER
1. **if** currPhase = MovingToTopPosition **and** TopPosition
2. **then** currPhase := ClosedForForging

(R_5) CLOSED
1. **if** currPhase = ClosedForForging **and** ForgingCompleted
2. **then** currPhase := MovingToBottomPosition

(R_6) MOVING TO LOWER
1. **if** currPhase = MovingToBottomPosition **and** BottomPosition
2. **then** currPhase := OpenForUnloading

According to the testing process discuss above, the ASM model will be translated into SMV. Observe that environmental variables, i.e. monitored variables that are not controlled by any of the agents in the system, have to become a part of the model to avoid their behaviour to remain unspecified in the resulting SMV model. We will not going into the technical details on how this is performed here. For more details, please refer to [Win].

The next step is to select the terms to be tested. For the purpose of this exercise, let us analyse TableLoaded in all the agents according to the **All-defs** criterion. The def, p-use, and c-use sets are generated according to the definitions described in section 3., and the following set of formulas is produced accordingly:

$$\{(wctl(d_{\mathsf{FB},R_3}^{\mathsf{TableLoaded},3}, p_{\mathsf{FB},R_1}^{\mathsf{TableLoaded},3}) \vee wctl(d_{\mathsf{FB},R_3}^{\mathsf{TableLoaded},3}, p_{\mathsf{FB},R_1}^{\mathsf{TableLoaded},5}) \vee wctl(d_{\mathsf{FB},R_3}^{\mathsf{TableLoaded},3}, p_{\mathsf{FB},R_2}^{\mathsf{TableLoaded},1}) \vee$$
$$wctl(d_{\mathsf{FB},R_3}^{\mathsf{TableLoaded},3}, p_{\mathsf{ERT},R_1}^{\mathsf{TableLoaded},1}) \vee wctl(d_{\mathsf{FB},R_3}^{\mathsf{TableLoaded},3}, p_{\mathsf{ERT},R_3}^{\mathsf{TableLoaded},1}) \vee wctl(d_{\mathsf{FB},R_3}^{\mathsf{TableLoaded},3}, p_{\mathsf{R},R_1}^{\mathsf{TableLoaded},1})),$$
$$(wctl(d_{\mathsf{R},R_2}^{\mathsf{TableLoaded},3}, p_{\mathsf{FB},R_1}^{\mathsf{TableLoaded},3}) \vee wctl(d_{\mathsf{R},R_2}^{\mathsf{TableLoaded},3}, p_{\mathsf{FB},R_1}^{\mathsf{TableLoaded},5}) \vee wctl(d_{\mathsf{R},R_2}^{\mathsf{TableLoaded},3}, p_{\mathsf{FB},R_2}^{\mathsf{TableLoaded},1}) \vee$$
$$wctl(d_{\mathsf{R},R_2}^{\mathsf{TableLoaded},3}, p_{\mathsf{ERT},R_1}^{\mathsf{TableLoaded},1}) \vee wctl(d_{\mathsf{R},R_2}^{\mathsf{TableLoaded},3}, p_{\mathsf{ERT},R_3}^{\mathsf{TableLoaded},1}) \vee wctl(d_{\mathsf{R},R_2}^{\mathsf{TableLoaded},3}, p_{\mathsf{R},R_1}^{\mathsf{TableLoaded},1}))\}$$

These formulas will be checked against the model and witnesses will be produced for them (if possible). We have performed this step manually, and found the following witness (test case) satisfying the above formulas (and therefore the chosen criterion). Observe that we do not report here the transition rules modeling the environment (**Env**).

TC1:

[FB] currPhase(FB) = CriticalRun, FeedBeltFree = false, TableLoaded = false [TableReadyForLoading=true]
[ERT] currPhase(ERT) = StoppedInLoadPosition
[TC] currPhase(TC) = WaitingToUnloadDepositBelt=true, PieceAtDepositBeltEnd = false
[R] currPhase(R) = WaitingInUnloadTablePos, PressLoaded = false
[TableReadyForUnloading = false, PressReadyForUnloading = false, PressReadyForLoading = false]
[DB] currPhase(DB) = NormalRun, DepositBeltReadyForLoading = false
[P] currPhase(P) = OpenForLoading,
[Env] PieceInFeedBeltLightBarrier = false, UnloadPositionReached = false, LoadPositionReached = false, UnloadingDepositBeltCompleted = false, LoadFeedBeltPosReached = false, LoadingFeedBeltComple eted = false, [UnloadPress|LoadDepBelt|LoadPress|UnloadTable]PosReached = false, [UnloadingTable |UnloadingPress|LoadingDepBelt|LoadingPress]Completed = false, PieceInDepositBeltLightBarrier = false, PieceAtDepositBeltEnd = false, MiddlePosition = false, TopPosition = false, ForgingCompleted = false, BottomPosition = false

For brevity here we will only show that this configuration triggers a run that satisfies the disjuncts

$$wctl(d_{\mathsf{FB},R_3}^{\mathsf{TableLoaded},3}, p_{\mathsf{ERT},R_1}^{\mathsf{TableLoaded},1}) =$$
$$\mathbf{EF}\ ((\mathsf{currPhase(FB)} = \mathsf{CriticalRun} \wedge \neg \mathsf{PieceInFeedBeltLightBarrier}) \wedge$$
$$\mathbf{EXE}\ (\neg\ (\mathsf{currPhase(R)} = \mathsf{UnloadingTable} \wedge \mathsf{UnloadingTableCompleted})\ \mathbf{U}$$

$$(\mathsf{currPhase(ERT)} = \mathsf{StoppedInLoadPosition} \wedge \mathsf{TableLoaded})))$$

$$wctl(d^{\mathsf{TableLoaded},3}_{\mathsf{FB},R_3}, p^{\mathsf{TableLoaded},3}_{\mathsf{FB},R_1}) =$$
$$\mathbf{EF}\ ((\mathsf{currPhase(FB)} = \mathsf{CriticalRun} \wedge \neg\mathsf{PieceInFeedBeltLightBarrier}) \wedge$$
$$\mathbf{EXE}\ (\neg\ (\mathsf{currPhase(R)} = \mathsf{UnloadingTable} \wedge \mathsf{UnloadingTableCompleted})\ \mathbf{U}$$
$$(\mathsf{currPhase(FB)} = \mathsf{NormalRun} \wedge \mathsf{PieceInFeedBeltLightBarrier})))$$

in the first formula which therefore holds true.

Running the system with this state, rule R_3 in **Feed Belt** fires, and as a consequence TableLoaded is in def_s in the new state:

[FB] currPhase(FB) = NormalRun, FeedBeltFree=false, TableLoaded=true, [TableReadyForLoading = true]
[ERT] currPhase(ERT) = StoppedInLoadPosition
[TC] currPhase(TC) = WaitingToUnloadDepositBelt=true, PieceAtDepositBeltEnd = false
[R] currPhase(R) = WaitingInUnloadTablePos, PressLoaded = false
[TableReadyForUnloading = false, PressReadyForUnloading = false, PressReadyForLoading = false]
[DB] currPhase(DB) = NormalRun, DepositBeltReadyForLoading = false
[P] currPhase(P) = OpenForLoading,
[Env] PieceInFeedBeltLightBarrier = true, UnloadPositionReached = false, LoadPositionReached = false, UnloadingDepositBeltCompleted = false, LoadFeedBeltPosReached = false, LoadingFeedBeltComple eted = false, [UnloadPress|LoadDepBelt|LoadPress|UnloadTable]PosReached = false, [UnloadingTable |UnloadingPress|LoadingDepBelt|LoadingPress]Completed = false, PieceInDepositBeltLightBarrier = false, PieceAtDepositBeltEnd = false, MiddlePosition = false, TopPosition = false, ForgingCompleted = false, BottomPosition = false

In this state, both rule R_1 in **Feed Belt**[6] and R_1 in **Elevating Rotary Table** are enabled to fire since their guards hold true. By definition, this means that TableLoaded is in p-use$_s$ w.r.t. $p^{\mathsf{TableLoaded},1}_{\mathsf{ERT},R_1}$ and $p^{\mathsf{TableLoaded},3}_{\mathsf{FB},R_1}$ in this state, and therefore this test case triggers a def-clear(TableLoaded) run exercising the du-pairs $(d^{\mathsf{TableLoaded},3}_{\mathsf{FB},R_3}, p^{\mathsf{TableLoaded},1}_{\mathsf{ERT},R_1})$ and $(d^{\mathsf{TableLoaded},3}_{\mathsf{FB},R_3}, p^{\mathsf{TableLoaded},3}_{\mathsf{FB},R_1})$ as desired.

Notice that actually this run uncovers an error in the specification: it violates the *Feed Belt Safety Property*—the feed belt does not put metal blanks on the table if the latter is already loaded or not stopped in loading position. This is due to an error in the definition of TableReadyForLoading which holds true even though TableLoaded is still true. The problem consists in the fact that the **or** operator was used in place of the **and** one in the defining equation of TableReadyForLoading.

7. Discussion and Future work

In the model-driven software engineering approach, a model is used to drive (or generate automatically) the code. Therefore, models are not used only as oracles to generate tests and considered correct by assumption, as done in many existing MBT techniques, but require a high degree of testing themselves.

Data flow coverage criteria can be used to bridge the gap between control flow testing and the ambitious and often unfeasible requirement to exercise every path in a program. Originally, they were developed for single modules in procedural languages [LK83, Nta84, RW85], but have since been extended for interprocedural programs in procedural languages [HS89], object-oriented programming languages [HR94], modeling languages such as UML [BLL05], and Web services applications [MCT08]. Tools to check the adequacy of test cases w.r.t data flow coverage criteria are being developed for programming languages such as Java (see for instance Coverlipse [Cov]).

In this paper we have significantly revised the theory and test case generation approach presented in [Cav08] and [Cav09] to amend some errors and imprecisions, and to deal with turbo ASMs; in particular, we have tackled the challenges introduced by the operator for sequential composition (**seq**). We have also presented a scenario of application of the proposed testing process to the Production Cell example.

[6] Observe that PieceInFeedBeltLightBarrier has now become true for effect of transition rules in **Env**.

We have illustrated a family of data flow coverage criteria for Abstract State Machines based on those introduced in [RW85]. We have explained why such criteria cannot be straightforwardly applied to ASMs, and have modified them accordingly. The criteria defined here focus on the interaction of portions of the ASM linked by the flow of data rather than merely by the flow of control. Therefore, they can also serve as a guide for a clever selection of critical paths for testing. We are not advocating that data flow coverage criteria should be applied necessarily to all the terms in an ASM model, but to a selection of critical terms. Moreover, we could restrict the application of coverage criteria to interesting subsets of agents. Finally, we have presented a model checking-based approach to generate automatically test suites satisfying the *all-defs* and *all-uses* criteria by formalising such criteria in temporal logic. Our approach builds on the work in [HCL$^+$03], which for this purpose uses CTL as temporal logic and SMV as model checker.

Other data flow coverage criteria, such as those proposed by Ntafos [Nta84] and Laski and Korel [LK83] do not seem to be adaptable to ASMs, as they are intrinsically linked to control flow graphs (they are strictly based on static analysis and observations of control flow graphs).

In general, when compared to the structural coverage criteria in [GR01], it is easy to see that the Rule Coverage Criterion is weaker than the all-uses and all-defs criteria: even though a test suite guarantees the execution of all the rules in the model at least once, it will not necessarily cover all the du-pairs for all the terms in the model. Consider the ASM in Example 2.1. The test suite

$[\mathcal{A}_1\ ]\ st(\mathcal{A}_1) = N,\ x = 6,\ y = 4$
$[\mathcal{A}_2\ ]\ st(\mathcal{A}_2) = N,\ z = 3,\ v = 1[w = 24]$

satisfies the Rule Coverage Criterion, but does not satisfies the all-uses (or even the all-defs) criterion as it never executes the assignment at line 3. and 4. in R_1 in the module of $\mathcal{A}_1$. Viceversa, a test suite that satisfy the all-uses criterion must execute all the rules in the model at least once (otherwise it would leave the assignments and predicates in the rule uncovered). Similarly, the rule update coverage criterion is weaker than the all-uses criterion. However, the more advanced criteria, i.e. the Parallel Rule Coverage and the Modified Condition Decision Coverage are not directly comparable with the all-uses criterion.

Observe that the main purpose of this work is to define a sound theory for data-flow testing ASMs. At the moment there is no complete tool support for the theory illustrated in this paper. In fact, although a formal mapping from ASMs to SMV has been defined [Win97], the interface developed in [CW00] is linked to the Workbench tool [Cas01] which unfortunately is not maintained anymore. However, there are plans to adapt it to work with the ASM tools currently available [FGG07, ASM]; this will allow us to develop a testing tool based on our approach and thus to evaluate its effectiveness and scalability by applying it to a number of case studies. We also intend to explore the possibility of adapting our data flow coverage criteria to work with the SPIN model checker, exploiting the ASM to PROMELA mapping defined in [GRR03, FGM07].

It would be interesting to include a component in the tool able to measure the coverage of a given test suite also with respect to control-flow criteria, and report exactly what definition-use pairs a given test manages to cover (typically, a test case will cover more than one per term). Moreover, we are interested in studying the problem of combining efficiently our data flow coverage criteria with the stronger structural criteria available.

References

[ASM] ASMETA: a tool set for the ASM. `http://sourceforge.net/projects/asmeta`.

[BCR00] Egon Börger, Alessandra Cavarra, and Elvinia Riccobene. Modeling the dynamics of UML state machines. In *Abstract State Machines*, volume 1912 of *Lecture Notes in Computer Science*, pages 223–241. Springer, 2000.

[BLL05] Lionel C. Briand, Yvan Labiche, and Q. Lin. Improving statechart testing criteria using data flow information. In *ISSRE*, pages 95–104, 2005.

[BM97] Egon Borger and Luca Mearelli. Integrating ASMs into the software development life cycle. *in J.UCS*, 3:603–665, 1997.

[BS00] Egon Börger and Joachim Schmid. Composition and submachine concepts for sequential asms. In *CSL*, volume 1862 of *Lecture Notes in Computer Science*, pages 41–60. Springer, 2000.

[BS03] Egon Börger and Robert F. Stärk. *Abstract State Machines. A Method for High-Level System Design and Analysis.* Springer, 2003.

[Cas01] Giuseppe Del Castillo. The ASM Workbench - A Tool Environment for Computer-Aided Analysis and Validation of Abstract State Machine Models Tool Demonstration. In *TACAS*, pages 578–581, 2001.

[Cav08] Alessandra Cavarra. Data flow analysis and testing of abstract state machines. In *ABZ*, volume 5238 of *LNCS*, pages 85–97. Springer-Verlag, 2008.

[Cav09] Alessandra Cavarra. Inter-agent data flow analysis of abstract state machines. In *Australian Software Engineering Conference (ASWEC)*. IEEE Computer Society, 2009.

[CES86] Edmund M. Clarke, E. Allen Emerson, and A. Prasad Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. Program. Lang. Syst.*, 8(2):244–263, 1986.

[CGMZ95] Edmund M. Clarke, Orna Grumberg, Kenneth L. McMillan, and Xudong Zhao. Efficient Generation of Counterexamples and Witnesses in Symbolic Model Checking. In *DAC*, pages 427–432, 1995.

[Cov] Coverlipse. `http://coverlipse.sourceforge.net/index.php`.

[CW00] Giuseppe Del Castillo and Kirsten Winter. Model Checking Support for the ASM High-Level Language. In *TACAS*, pages 331–346, 2000.

[FGG07] Roozbeh Farahbod, Vincenzo Gervasi, and Uwe Glässer. CoreASM: An Extensible ASM Execution Engine. *Fundam. Inform.*, 77(1-2):71–103, 2007.

[FGM07] Roozbeh Farahbod, Uwe Glsser, and George Ma. Model Checking CoreASM Specifications. In *Proc. 14th Intl. Workshop on Abstract State Machines. Agder University College. Faculty of Engineering and Science. Grimstad, Norway. ISBN 978-82-7117-627-3*, 2007.

[FW88] Phyllis G. Frankl and Elaine J. Weyuker. An applicable family of data flow testing criteria. *IEEE Trans. Software Eng.*, 14(10):1483–1498, 1988.

[FW93] Phyllis G. Frankl and Elaine J. Weyuker. A formal analysis of the fault-detecting ability of testing methods. *IEEE Trans. Software Eng.*, 19(3):202–213, 1993.

[GR01] Angelo Gargantini and Elvinia Riccobene. ASM-Based Testing: Coverage Criteria and Automatic Test Sequence. *J. UCS*, 7(11):1050–1067, 2001.

[GRR03] Angelo Gargantini, Elvinia Riccobene, and Salvatore Rinzivillo. Using Spin to Generate Tests from ASM Specifications. In *Abstract State Machines*, pages 263–277, 2003.

[Gur95] Y. Gurevich. *Specification and Validation Methods. In Evolving Algebras 1993: Lipari Guide*. Oxford University Press, 1995.

[HCL+03] Hyoung Seok Hong, Sung Deok Cha, Insup Lee, Oleg Sokolsky, and Hasan Ural. Data Flow Testing as Model Checking. In *ICSE*, pages 232–243, 2003.

[Hol97] Gerard J. Holzmann. The model checker SPIN. *Software Engineering*, 23(5):279–295, 1997.

[HR94] Mary Jean Harrold and Gregg Rothermel. Performing data flow testing on classes. In *SIGSOFT '94: Proceedings of the 2nd ACM SIGSOFT symposium on Foundations of software engineering*, pages 154–163. ACM, 1994.

[HS89] Mary Jean Harrold and Mary Lou Soffa. Interprocedual data flow testing. In *TAV3: Proceedings of the ACM SIGSOFT '89 third symposium on Software testing, analysis, and verification*, pages 158–167. ACM, 1989.

[ITU00] ITU-T. SDL formal semantics definition. ITU-T Recommendation Z.100 Annex F, International Telecommunication Union, November 2000.

[LK83] Janusz W. Laski and Bogdan Korel. A data flow oriented program testing strategy. *IEEE Trans. Software Eng.*, 9(3):347–354, 1983.

[McM93] Kenneth L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.

[MCT08] Lijun Mei, W. K. Chan, and T. H. Tse. Data flow testing of service-oriented workflow applications. In *International Conference on Software Engineering (ICSE 2008), Leipzig, Germany*, pages 371–380. ACM, 2008.

[Nta84] Simeon C. Ntafos. On required element testing. *IEEE Trans. Software Eng.*, 10(6):795–803, 1984.

[RW85] Sandra Rapps and Elaine J. Weyuker. Selecting software test data using data flow information. *IEEE Trans. Software Eng.*, 11(4):367–375, 1985.

[SSB01] Robert F. Stärk, Joachim Schmid, and Egon Börger. *Java and the Java Virtual Machine: Definition, Verification, Validation*. Springer, 2001.

[UPL06] Mark Utting, Alexander Pretschner, and Bruno Legeard. A Taxonomy of Model-Based Testing. In *Working paper series. University of Waikato, Department of Computer Science. No. 04/2006. New Zealand*, 2006.

[Wey93] Elaine J. Weyuker. More experience with data flow testing. *IEEE Trans. Software Eng.*, 19(9):912–919, 1993.

[Win] Kirsten Winter. *Model Checking for Abstract State Machines*. PhD thesis, Technischen Universitaet Berlin, 2001.

[Win97] Kirsten Winter. Model Checking for Abstract State Machines. *J. UCS*, 3(5):689–701, 1997.